Universidade Estadual de Campinas
Faculdade de Engenharia Elétrica e de Computação
Departamento de Engenharia de Computação e Automação Industrial

Gestão de Configuração para Teste de Software

André Luiz de Castro Villas Boas

Orientador: **Prof. Dr. José Carlos Maldonado**
Co-orientador: **Prof. Dr. Mario Jino**

Dissertação apresentada à Faculdade de Engenharia Elétrica e de Computação da Universidade Estadual de Campinas como parte dos requisitos exigidos para a obtenção do título de mestre em Engenharia Elétrica.

Campinas - SP - Brasil
Julho de 2003

FICHA CATALOGRÁFICA ELABORADA PELA
BIBLIOTECA DA ÁREA DE ENGENHARIA - BAE - UNICAMP

V713g

Villas Boas, André Luiz de Castro
Gestão de configuração para teste de software / André Luiz de Castro Villas Boas.--Campinas, SP: [s.n.], 2003.

Orientadores: José Carlos Maldonado e Mario Jino.
Dissertação (mestrado) - Universidade Estadual de Campinas, Faculdade de Engenharia Elétrica e de Computação.

1. Engenharia de software. 2. Software - Validação. 3. Gerenciamento de configurações de software. I. Maldonado, José Carlos. II. Jino, Mario. III. Universidade Estadual de Campinas. Faculdade de Engenharia Elétrica e de Computação. IV. Título.

RESUMO

Este trabalho propõe o uso de conceitos de gestão de configuração de software para apoiar a atividade de teste (sendo a parte experimental focada no teste de unidade), com o objetivo de organizar o ambiente de teste de tal forma que o testador tenha os dados de teste disponíveis para uma possível evolução do software. O problema aqui tratado está baseado em dois fatos principais: o primeiro é que, em geral, a atividade de teste de unidade é feita de forma pouco sistemática e quase nunca documentada; o segundo é que, mesmo com a ajuda de ferramentas que automatizam parte das atividades de teste, inúmeras vezes a execução dos casos de teste e sua posterior análise são atividades custosas, ou às vezes inviáveis. Num primeiro momento foi caracterizado um item de configuração de teste (ICT), baseado na importância e no custo de se obter esse determinado item. Com isso, dada uma ferramenta de teste e estudando-se seu comportamento, pode-se claramente determinar o conjunto de informações que forma o ICT para essa ferramenta. Num segundo momento, partiu-se para a automatização do processo de coleta, proteção e recuperação da informação de teste, com o uso de uma ferramenta de controle de configuração. Um ambiente que implementa a gestão de configuração de software para teste, completamente independente da ferramenta de teste usada, foi desenvolvido usando como base os comandos RCS.

ABSTRACT

This work proposes the usage of software configuration management concepts to support the test activity (taking into accounto that the experimental part has focused the unit test), aiming to organize the test environment, so that test data be available to the tester, making possible to evolve the software. The problem raised is based in two mainly issues: first, generaly the test activity is not carried out in a sistematic manner end almost is never documented; second, even with the help of some test tools, that are capable to automatize part of the test activities, many times the test execution and its later analisis are costly activities, or some times not feasible. To solve this a test configuration iten (TCI) has been characterized, based on the importance and the costs to get it. With this rationale it is possible to determine the TCI for any test tool. After that, the test information collect, protect and recover process has been automatized with the use of a software configuration control tool. An evironment that implement the software configuration management for test, completely independent from the test tool has been developed using the RCS commands.

Eu gostaria de dedicar esse trabalho à memória de minha mãe, que infelizmente se foi antes de vê-lo apresentado. Mas se hoje ele se parece com um texto em português é porque com uma paciência que era só dela, ela o revisou inteiramente antes de nos deixar.

Mãe, é difícil dizer qualquer coisa nesse momento, até porque a expressão "muito obrigado" não consegue traduzir tudo o que você fez por nós três e em particular por mim. Foram anos maravihosos, de alegria, de aprendizado, de carinho, de dedicação, de amor e de bons exemplos. Exemplos que nos educaram muito mais do que palavras vazias.

Hoje, olhando para trás em minha vida, eu não consigo ver além do meu nascimento, mas em função de tudo que pudemos partilhar posso dizer com certeza de que te escolhi como minha mãe e não poderia ter sido melhor. Além de mãe, você foi minha amiga, minha conselheira, meu porto seguro. E se hoje, depois da tua partida eu não me perdi, é porque aprendi muito bem as tuas lições que nos ensinaram a vencer todos os obstáculos da vida. Um dia você ainda vai me dizer como fui como filho, mas eu posso te dizer já que você foi muito mais do que eu até mesmo merecia. Nosso amor nos manterá eternamente juntos.

AGRADECIMENTOS

Agradeço imensamente aos meus pais, João Ignácio e Maria Rute (*in memoriam*), que nunca mediram esforços para me dar três coisas: amor, disciplina e educação formal. Tudo que tenho e sou hoje, devo a eles. A meus irmãos, João Paulo e Fabiana, meus companheiros de jornada nessa vida, muito obrigado pelo eterno carinho e apoio.

À minha esposa Márcia, que com uma paciência incrível tolerou todos os momentos que subtraí de nosso convívio para executar esse trabalho e ainda assim me impulsionava para frente. Obrigado meu amor, teu apoio e carinho foram fundamentais para meu sucesso.

Aos meus orientadores Prof. José Carlos Maldonado e Prof. Mario Jino pela atenção, amizade e profissionalismo com que me guiaram nessa empreitada. Obrigado por tudo. Mais do que orientadores, eu encontrei dois amigos que sempre serão exemplo para a minha formação.

Aos amigos do antigo projeto ASSISTE do CPqD de onde nasceu a idéia desse trabalho, quando eu ainda estava engatinhando na seara da qualidade de software. Obrigado pelos anos maravilhosos que trabalhamos juntos.

À minha prima Valquíria e meu amigo Fábio, que em todos os momentos estiveram me empurrando para frente para que eu finalizasse esse trabalho. Muito obrigado pelo apoio.

Ao amigo Edésio por todas as dicas e dúvidas que ele me tirou sobre o uso dos *scripts*. Obrigado pela paciencia com as minhas dúvidas.

À Tereza e ao Furia, meu muito obrigado pela inestimável ajuda no uso do ambiente FrameMaker. Eu sei que eu dei muito trabalho a vocês.

Aos meus amigos da UNICAMP e da USP/SCar, Auri, Plínio, Dino, Paulinho, Delamaro, Adenilson, Chaim e tantos outros que com certeza vou me esquecer de colocar aqui. Obrigado pela amizade e pelos empurrões.

Índice

Capítulo 1

Capítulo 2

Capítulo 3

Capítulo 5

Capítulo 6

Índice de Figuras

Índice das Tabelas

Capítulo 1

Introdução

1.1 Contexto e motivação

Identifica-se nos últimos anos uma forte tendência à busca contínua pela qualidade, principalmente porque o resultado de métodos que objetivam a melhora dessa qualidade pôde ser traduzido em números, ou seja, foi percebido como lucro para a empresa (diminuição de retrabalho, maior produtividade, uso racional de recursos, etc). Isto pode ser visto nos vários indicadores apresentados por Caper Jones[42], ou nas melhoras sobre prazo de entrega e ciclo de reteste apresentados pela Schlumberger[69], ou ainda no Retorno Sobre o Investimento apresentado pela Rayheon[23]. Essa também é uma preocupação da Diretoria de Sistemas de Suporte à Operação (DSSO) da Fundação CPqD, que busca qualidade nos projetos sob sua orientação como ponto de diferenciação de seus produtos num mercado aberto e competitivo.

Uma das atividades de garantia da qualidade de um produto de software é o teste, para certificar-se de sua aderência aos requisitos especificados. Teste e Validação de software é uma das áreas de pesquisa da Engenharia de Software e constitui um dos elementos para aprimorar a produtividade e fornecer evidências da confiabilidade do software. No processo de teste, as atividades de projeto, execução e avaliação dos casos de teste são de extrema importância. Para tanto utiliza-se um conjunto de técnicas, métodos e critérios, teoricamente fundamentados que sistematizam essas atividades[47]. Mas, embora muito comentadas, elas ainda não são profundamente utilizadas de forma sistemática pela maioria dos profissionais de software. Inúmeros programadores, quando questionados, respondem que testam seus programas para verificar se eles funcionam. Não deixa de ser uma resposta, embora muito simplificada para a extensão da questão. Na pesquisa realizada pela Secretaria de Política de Informática do Ministério da Ciência e Tecnologia (MCT/SEPIN)[59] constata-se que apenas 37% das empresas planejam as atividades de teste e 23% fazem uso da discilplina de gestão de configuração (embora a mesma pesquisa aponte que apenas 20% fazem uso de uma ferramenta para gestão de configuração, 2,6% usem analisadores de cobertura de código e 4,5% usem ferramenta de *capture&play-back* para teste).

No dicionário Aurélio[24] encontra-se para o verbete **teste**: *"S. m. 1. Exame, verificação ou prova para determinar a qualidade, a natureza ou o comportamento de alguma coisa, ou de um sistema sob certas condições. 2. Método, processo, procedimento ou meios utilizados para tal exame, verificação ou prova."* A palavra teste deriva da palavra inglesa *test*, que por sua vez tem origem na palavra latina *testum*, que representa um pote de barro utilizado para determinar a presença ou medir o peso de vários elementos[28]. Hoje em dia, ela é largamente utilizada e praticamente

qualquer pessoa tem uma idéia de seu significado, principalmente se levarmos em consideração que todos fizemos, um dia, *testes* na escola.

No entanto, o conceito de teste em computação não é tão intuitivo e por isso, durante muito tempo, esta atividade não foi encarada de forma sistemática. Foi somente após a realização da Primeira Conferência Formal em Teste de Software (na Universidade da Carolina do Norte, 1972) que a disciplina de teste ganhou caráter formal e várias outras conferências foram realizadas, assim como vários cientistas começaram a se dedicar a este assunto. Sobre a definição de teste, podemos citar alguns autores como Hetzel[29] , **"Testar é o processo de certificar que o programa faz o que era suposto fazer"** ou Glen Myers[51], **"Testar é o processo de executar um programa ou sistema com o objetivo de encontrar erros"** e veremos enfoques diferentes, mas complementares.

Neste trabalho, teste deve ser entendido segundo Myers, ou seja, como o ato de executar um programa, fornecendo dados de entrada, verificando os resultados obtidos e comparando-os com os resultados esperados, com o objetivo de revelar a presença de defeitos, ou erros (não faremos aqui distinção desses termos). Com relação ao termo programa, ele é aqui utilizado no sentido de um procedimento ou função escrito em alguma linguagem de programação. Outros termos que também são utilizados com o mesmo sentido são: unidade ou módulo.

Boris Beizer[4] explica que erros cometidos durante o desenvolvimento são causados pelo fato do ser humano possuir capacidade limitada de tratar problemas complexos e de comunicar-se de forma perfeita, sem equívocos de interpretação. Mesmo quando se utilizam técnicas mais rigorosas para o desenvolvimento de software (como as baseadas em semântica formal e métodos de provas formais), como uma forma de gerar código livre (ou quase livre) de erro, os sistemas de software ainda são liberados com erros. Numa experiência realizada pela Praxis e relatada por Pfleeger e Hatton[56] foi levantado que o número de erros encontrados no código não apresentou grandes diferenças entre módulos formalmente especificados e os especificados de modo informal. Contudo, os dois autores fazem uma observação muito interessante dizendo: "Além do mais, métodos formais podem ser mais eficientes agindo como catalisadores para outras técnicas, especialmente para teste, devido à possibilidade de produção de componentes testáveis, ou pelo fornecimento de uma estrutura que serve de base para um teste de sistema mais abrangente".

Segundo Pressman[57] "O teste de software é um elemento crítico da garantia de qualidade de software e representa a última revisão da especificação, do projeto e da codificação". Por isso, o processo de teste deve ser executado durante todo o ciclo de vida do projeto do software. Do ponto de vista de custo, segundo Meyers[51], aproximadamente 50% do tempo e mais de 50% do custo total de desenvolvimento de um produto de software é gasto em teste; Harrold indica que

nos casos de software crítico essa porcentagem é ainda maior[27]. Pressman[57] diz que 40% do esforço de desenvolvimento é gasto em teste e que em projetos de software crítico (por exemplo, controle de vôo ou monitoramento de reatores nucleares) o seu custo pode chegar a três vezes o custo de todas as outras fases de desenvolvimento. O uso de informações de teste para redução do custo da atividade de teste de regressão já está sendo explorado, como pode ser visto em [70]. Essas informações são muito importantes para se lidar com alterações de forma organizada.

A atividade de teste é usualmente conduzida em fases conhecidas como: teste de unidade, teste de integração, teste de sistema e teste de aceitação. Um produto de software pode ser testado utilizando-se algumas técnicas, sendo três as mais usadas[47]: Teste Funcional ou Caixa-Preta; Teste Estrutural ou Caixa-Branca e Teste baseado em erros, que serão melhor discutidas neste trabalho.

A atividade de teste em si pode ser vista como um desenvolvimento em paralelo, já que necessita de um planejamento, da geração de casos de teste, da execução desses testes, do armazenamento dos resultados e do julgamento sobre a confrontação dos resultados esperados contra os obtidos. A qualidade e produtividade dessa atividade são fortemente dependentes do critério utilizado e da existência de ferramental de apoio, já que sem ferramentas a atividade de teste se torna uma atividade propensa a erros e limitada a programas muito simples. As ferramentas também servem como catalisadores no ensino da disciplina de teste de software a pesquisadores e alunos, bem como permite a transferência de tecnologia para as indústrias. Essa possibilidade de tratar programas mais complexos faz com que o número de elementos manuseados seja muito elevado em projetos de grande porte. Daí a necessidade de, também, haver toda uma preocupação com a qualidade da atividade de teste, ou seja, com a maneira pela qual pode-se geri-la de modo mais eficaz e mais garantido. Essa necessidade também foi sentida durante a execução do "Projeto Validação e Teste de Sistemas de Operação - PVTSO"[19], realizado num convênio entre CPqD, USP/São Carlos e UNICAMP e que também serviu de motivador para este trabalho. A solução apresentada neste trabalho baseia-se na idéia de utilizar fundamentos da gestão de configuração para auxiliar no controle do processo de teste e de todos os elementos envolvidos nele. Na verdade, a configuração de teste deve ser parte da configuração do produto de software. Mas para que serve, afinal, a gestão de configuração?

Bersoff[7] garante que a Gestão de Configuração nasceu como resposta aos erros cometidos durante os anos 70, quando computadores começaram a ser utilizados de maneira geral para a solução dos mais diversos, e complexos, problemas e para os quais o gerenciamento tradicional mostrou-se inadequado. Principalmente, quando se descobriu que a complexidade de gerenciamento de um sistema não variava de maneira linear com o número de linhas de código, mas de maneira exponencial. Como observa Babich, ela é uma disciplina útil para a gerência do projeto, pois lhe permite executar tarefas complexas de uma maneira mais organizada, o que proporciona

aumento de produtividade e redução de erros[57]. Do ponto de vista da atividade de teste, o uso da gestão de configuração como elemento de melhoria de qualidade ainda é uma questão em aberto. Mas ela é profundamente importante no que tange aos benefícios trazidos por ela, pois o tipo e quantidade de informação advindas da atividade de teste que podem ser manipuladas com o uso dessa disciplina, possibilitam maior eficiência nas atividades de evolução, depuração e manutenção do software. Essa grande quantidade de informação também levanta a necessidade de que a atividade de teste seja bem documentada e de que modelos de documentos para apoio ao teste sejam desenvolvidos (por exemplo, modelo de um plano de teste, de um relatório de problemas etc.). Nesse sentido podemos encontrar alguns esforços de padronização de documentação para teste, como os do IEEE[33] ou exemplos encontrados em livros de teste, como os de Perry[55].

Com relação à melhoria de processos, Gestão de Configuração é disciplina básica para se ascender ao nível 2 do modelo de maturidade CMM/SEI[54] e o teste sistemático é uma atividade essencial para a ascenção ao nível 3 do modelo. Do ponto de vista da avaliação da qualidade de produtos de software usando a norma ISO 9126[38], o Processo para o Avaliador - ISO 14598-5[39] - estabelece que para a avaliação da característica **funcionalidade** são recomendadas as técnicas de teste funcional e teste estrutural.

1.2 Objetivo

O objetivo deste trabalho é propor o uso da gestão de configuração de software no contexto das atividades de teste de unidade, com o intuito de minimizar o esforço alocado à atividade de teste no desenvolvimento e na evolução do software. Nessa perspectiva, o objetivo é apresentar uma proposta para a caracterização de um item de configuração de teste e definir um ambiente que automatize a coleta, a recuperação e o controle desses itens de configuração de teste. O ambiente de teste definido é constituído por uma ferramenta de gestão de configuração desenvolvida com base nos comandos da ferramenta de controle de versões RCS[63] do UNIX integrada com uma ferramenta de teste. Para efeito do experimento aqui a ferramenta utilizada foi a POKE-TOOL[16] desenvolvida na UNICAMP e que apóia a aplicação de critérios baseados em Fluxo de Controle e em Fluxo de Dados.

1.3 Organização

Os capítulos subseqüentes desta dissertação estão assim organizados: no Capítulo 2, está descrito um dos ambientes de desenvolvimento de software do CPqD. No Capítulo 3, discutem-se a atividade de teste de software, técnicas e critérios associados e documentação associada ao processo de teste; também é tratada a gestão de configuração de software e sua relevância na atividade de teste. No Capítulo 4, apresentam-se a problemática do volume de dados gerado na atividade de teste e a gestão de configuração como uma maneira de controlar esses dados de forma ordenada.

Discutem-se ainda a caracterização de um item de configuração de teste e uma forma de apoiar a obtenção, a recuperação e o controle dos dados necessários de forma automática. No Capítulo 5 são apresentadas as ferramentas utilizadas, o ambiente implementado e um exemplo de aplicação desse ambiente de gestão de configuração de software para teste, proposto no Capítulo 4. Finalmente, no Capítulo 6, discutem-se as conclusões e possíveis desdobramentos deste trabalho. Nos Anexos de A a G são apresentados modelos de uma família de documentos para apoiar a atividade de teste, os quais foram baseados nas normas IEEE[33]. No Anexo H são apresentados os comandos implementados para o ambiente de gestão de configuração de software para teste. No Anexo I é apresentada a listagem do programa **`prime.c`** utilizado no exemplo.

Capítulo 2

Ambiente de Desenvolvimento do CPqD

2.1 Considerações Iniciais

O presente capítulo tem por objetivo tratar de um dos ambientes de desenvolvimento do CPqD. São introduzidos os sistemas de telecomunicações, com ênfase nos sistemas de operação. É apresentada uma das metodologias de desenvolvimento utilizadas, como é o processo de teste nessa metodologia e qual é o ambiente de desenvolvimento de software propriamente dito.

2.2 Os Sistemas de Suporte à Operação

Os Sistemas de Telecomunicações têm se caracterizado por um crescente aumento na sofisticação e complexidade das redes, como também por um aumento no uso da informática para apoiar sua administração. Esses sistemas são divididos em várias famílias, como sistemas de comutação, sistemas de transmissão, sistemas de comunicação celular, sistemas de gerência de redes e sistemas de operação. O software é elemento fundamental em todos eles.

Uma empresa de telecomunicações, para se manter competitiva no mercado atual, precisa proporcionar: quantidade crescente de melhores serviços, rapidez no fornecimento desses serviços e bom nível de qualidade no seu serviço de manutenção, além de agilidade nos seus processos burocráticos internos. Os Sistemas de Suporte à Operação, ou simplesmente Sistemas de Operação[68], são os elementos que garantem a uma Empresa Operadora de Telecomunicações a automatização da administração de sua rede de telecomunicações, tornando possível uma melhoria da qualidade dos serviços prestados, um aumento da produtividade e uma diminuição dos seus custos operacionais. Podemos citar como exemplos desses sistemas, o sistema de tarifação ou o sistema de tratamento de clientes.

A atuação do CPqD nesta área foi iniciada em 1989, após um processo de reavaliação do seu perfil de atuação no cenário tecnológico que culminou com a criação de uma unidade encarregada desses sistemas (a atual Diretoria de Sistemas de Suporte a Operações). Dentre os sistemas em desenvolvimento, três seguem o modelo de desenvolvimento aqui apresentado[1][71]:

- *SAGRE* - Sistema Automatizado de Gerenciamento de Rede Externa

1. Esses são os nomes antigos dos sistemas, mas o autor preferiu utilizá-los para manter a coerência com a nomenclatura utilizada na referência bibliográfica.

Tem como objetivo automatizar os processos relacionados com o cadastro, planejamento, projeto, construção, operação e manutenção da rede externa (que compreende todo o equipamento externo às centrais, como postes, cabos, caixas de passagem etc.).

- *SGE* - Sistema de Gerência de Equipamentos

Tem como objetivo automatizar os processos de cadastro, operação e manutenção da planta de equipamentos, dando também suporte à sua administração e planejamento e ao acompanhamento de ordens de serviço (como um pedido de manutenção, por exemplo).

- *Terus* - Sistema de Administração de Acomodação e Números

Tem como objetivo automatizar os procedimentos de designação de números e de faci-lidades em centrais de comutação (por exemplo, discagem abreviada) de forma a obter o balanceamento de tráfego entre diversos órgãos de uma central telefônica.

As características principais de desenvolvimento desses sistemas são o uso de:

- plataformas comerciais,
- sistema operacional UNIX,
- linguagens C/ANSI e SQL/ANSI,
- banco de dados relacional comercial,
- estações de trabalho e
- interfaces gráficas.

2.3 A Metodologia

A metodologia de desenvolvimento utilizada para alguns dos sistemas desenvolvidos no CPqD denomina-se MUSA [52]. Ela nasceu da necessidade de se padronizar o desenvolvimento de software no STB (antigo Sistema TELEBRÁS) com o objetivo de facilitar o intercâmbio e remanejamento dos sistemas de informação, bem como fornecer uma maior transparência, portabilidade e não dependência com relação aos indivíduos envolvidos no desenvolvimento desses sistemas.

O modelo de desenvolvimento utilizado pela MUSA é o clássico em cascata, composto de fases, as quais são divididas em etapas que são constituídas por atividades. A seguir, apresenta-se um resumo dessas fases. Maiores informações sobre a metodologia podem ser obtidas em [52].

1. ESPECIFICAÇÃO DE REQUISITOS:

Na especificação de requisitos é definido o problema a ser tratado. Aqui, analisa-se a situação atual, faz-se o levantamento dos requisitos do sistema, dos custos, dos recursos necessários e o cronograma inicial.

2. PROJETO LÓGICO:

No projeto lógico é definido como o problema será resolvido. Aqui, definem-se as funções que o sistema deve ter e o modelo de dados a ser seguido. Também se faz uma revisão dos recursos, prazos e custos.

3. PROJETO FÍSICO:

No projeto físico é estruturada a solução do problema. Aqui, definem-se os módulos do sistema, sua arquitetura, suas entradas e saídas e o modelo físico da base de dados. As funções do sistema são estruturadas hierarquicamente e agrupadas por tipo de processamento e ambiente operacional, caracterizando assim as Unidade de Implementação (UI). A UI é o menor nível de abstração da especificação; diferente da unidade, que é um simples componente de código, a UI é uma coleção destes componentes com uma funcionalidade bem definida. Nessa fase também é gerado um documento chamado "Especificação de Características de UI", no qual são especificadas as unidades componentes de cada UI. As interfaces entre o sistema e outros sistemas e as interfaces entre as várias UIs são definidas num documento chamado "Especificação de Definição das Interfaces". Também se faz uma revisão dos recursos, prazos e custos.

4. IMPLEMENTAÇÃO:

A implementação é a construção da solução do problema. Aqui, especificam-se e implementam-se os programas (as unidades componentes das UIs), realizam-se todos os testes, elaboram-se os procedimentos de produção e os de usuário.

5. IMPLANTAÇÃO:

Na implantação o sistema é instalado no ambiente alvo. Nesta fase, aprova-se a infra-estrutura operacional, realiza-se o treinamento dos usuários e coloca-se o sistema em operação em condições reais.

Como pode ser observado, o teste é parte da etapa de implementação e na prática observa-se que a atividade de teste formal é o teste de sistema, executado por um grupo à parte e constituindo a atividade de garantia da qualidade do produto de software desenvolvido, tanto o teste de unidade quanto o de integração são, na maioria das vezes, atividades informais. No entanto, Humphrey[31]garante que quando os módulos não são bem testados unitariamente, o teste de integração (e o de sistema, por conseqüência) pode sofrer grandes atrasos devido às paralisações necessárias para a correção dos módulos individuais. Tomando isto em consideração, pretende-se apresentar uma maneira de apoiar o teste de unidade e, com isso, torná-lo um atividade sistemática que auxilie na introdução de mecanismos de garantia da qualidade no desenvolvimento, já no início da fase de implementação e não apenas no seu final. Lembrando que como o ambiente de controle é aberto, ele poderia facilmente ser estendido para atender, também, ao teste de integração com o uso de ferramentas de teste que apóiem essa fase de teste[67]. Dessa forma, encaminha-se para a visão de Schach[58], de que a garantia de que está se fazendo certo o produto certo, deve ser diluída em atividades que acompanhem todo o ciclo de construção do software e não apenas possuir um único ponto no qual a qualidade do produto é examinada. E com isso, esse ambiente pode até mesmo contribuir para a melhoria da própria metodologia.

2.4 O Processo de Teste

O Processo de Teste é caracterizado por um conjunto de atividades que são executadas ao longo de todo o ciclo de desenvolvimento do software. Essas atividades estão agrupadas em etapas bem definidas, que são:

1. Etapa de planejamento:

Planejamento é a distribuição racional no tempo dos recursos disponíveis para a realização de alguma atividade. Planejar é, de modo geral, decidir antecipadamente o que deve ser feito e quando deve ser feito, ou seja, é traçar uma linha de ação. É justamente nesta etapa que será descrito o escopo dos testes, identificando métodos que serão empregados, recursos necessários, cronograma de atividades, pessoal necessário, itens que serão testados, itens que não serão testados, características dos itens testados e responsabilidades. Como resultado final desta etapa, teremos o documento **Plano de Teste (PN)**, cujo modelo é apresentado no Anexo A.

2. Etapa de projeto:

Projetar é especificar de maneira detalhada a forma de se realizar algo. Nesta etapa, será escolhido um grupo específico de características a serem testadas, descrevendo detalhadamente os métodos que serão empregados e os testes que deverão ser feitos. O resultado final desta etapa é o documento **Projeto de Teste (PJ)**, cujo modelo é apresentado no Anexo B.

3. Etapa de implementação:

Implementar é fornecer os subsídios indispensáveis à execução de alguma tarefa. Na etapa de implementação é que se descrevem detalhadamente os itens que serão testados, as entradas utilizadas, as saídas esperadas e os passos necessários para se executar os testes, ou seja, a seqüência de tarefas necessária para se analisar um item de software com a finalidade de se avaliar um conjunto de suas características. Teremos como resultado final desta etapa os documentos **Caso-de-Teste (CS)** e **Procedimento de Teste (PR)**, cujos modelos são apresentados, respectivamente, nos Anexos C e D.

4. Etapa de execução:

Esta etapa é a parte prática do teste, isto é, aquela em que se efetiva a tarefa de teste propriamente dita. Aqui, a partir do procedimento de teste que será seguido e dos casos-de-teste associados, o testador realizará os testes. Nesta etapa, ele deve registrar toda a atividade de teste, identificando data e hora do teste, autor do teste, o procedimento seguido, o pessoal envolvido, os resultados obtidos, as condições ambientais e os eventos não esperados (quando ocorrerem). Caso aconteçam problemas durante os testes, estes deverão ser relatados, sendo referidos o procedimento em uso, os casos-de-teste associados e os itens que estão sendo testados, fornecendo uma descrição dos problemas, as entradas utilizadas, data e hora do ocorrido, o passo do procedimento, os observadores e impacto do problema sobre os testes. Ao final das atividades de teste, um resumo deve ser fornecido referindo os itens testados, sumarizando os resultados obtidos e fornecendo uma avaliação baseada nesses resultados. Os produtos gerados nesta etapa são os documentos **Registro de Teste (RG)**, **Relatório de Problemas de Teste (RT)** e **Relatório de Resumo de Teste (RE)**, cujos modelos são apresentados, respectivamente, nos Anexos E, F e G.

Embora essas etapas ocorram em seqüência, nem sempre elas acontecem em bloco para cada teste; na verdade, elas são diluídas ao longo do ciclo-de-vida do projeto.

Após a fase de Especificação de Requisitos (já na fase de Projeto Lógico), a partir do documento onde estão especificados os requisitos do sistema a ser construído, um Plano de Teste de Sistema deve ser gerado. Com base nesse plano, são gerados os Projetos de Teste e Casos-de-Teste de Sistema.

Durante a fase de Projeto Físico, com base no documento de Especificação de Características de UI, os Plano, Projetos e Casos-de-Teste de UI devem ser gerados. E, baseado no documento de Especificação de Definição das Interfaces, os Plano, Projetos e Casos-de-Teste de Integração serão gerados.

Na fase de Implementação, são gerados os documentos Procedimentos de Teste (sistema, integração e UI). Neste momento, cada programador, ao finalizar a implementação do software que está sob sua responsabilidade, executa o Teste de Unidade para seu software. A partir de grupos de unidades corretas são iniciados os Testes de UI. Caso seja encontrado algum problema nesse teste, as unidades que apresentaram defeito são devolvidas para a equipe de desenvolvimento (com os relatórios de problemas) para serem corrigidas e testadas novamente (teste de unidade). Quando um grupo de UIs que compõem um subsistema passa pelo Teste de UI, pode ser iniciado o Teste de Integração (que em caso de problema sofre o mesmo procedimento descrito acima). E quando os Testes de Integração terminam, pode-se executar o Teste de Sistema.

2.5 Documentação Gerada no Processo de Teste

Na etapa de teste, durante o ciclo de vida do projeto, são gerados uma série de documentos que servem para acompanhamento e controle do projeto. Na versão original da metodologia MUSA essa documentação não existia. Existem vários modelos de documentos de teste sendo hoje utilizados. Aqui, foi proposta uma documentação baseada no padrão de documentos de teste ANSI/IEEE[35], que até hoje é utilizada com as devidas alterações que a experiência de uso trouxe (por exemplo, simplificação de alguns itens, aglutinação de dois documentos em um etc.). Nos anexos de A a G, os documentos (e suas estruturas) são descritos na seguinte seqüência:

- **Anexo A:** *Plano de Teste*

Descreve o escopo, os métodos, os recursos e o cronograma das atividades de teste. Identifica os itens que serão testados, as características a serem testadas, as tarefas de teste que serão executadas, o pessoal responsável por essas tarefas e os riscos associados ao plano.

- **Anexo B:** *Projeto de Teste*

Especifica, de maneira detalhada, o(s) método(s) de teste e identifica as características que serão testadas por este projeto, bem como os testes associados a ele.

- **Anexo C:** *Caso de Teste*

Especifica um caso de teste identificado por um Projeto de Teste. Descreve detalhadamente os itens que serão testados, as entradas necessárias e as saídas esperadas.

- **Anexo D**: *Procedimento de Teste*

Especifica os passos necessários para executar um conjunto de casos de teste.

- **Anexo E**: *Registro de Teste*

Fornece um registro cronológico de detalhes relevantes à execução dos testes. Indica qual procedimento de teste está sendo executado e relaciona (caso haja) todos os Relatórios de Problemas de Teste gerados.

- **Anexo F:** *Relatório de Problemas de Teste*

Documenta qualquer evento ocorrido no processo de teste e que mereça ser investigado. Deve indicar os itens de teste envolvidos, o procedimento de teste executado, os casos de teste utilizados e o registro de teste relacionado.

- **Anexo G:** *Relatório de Resumo de Teste*

Resume os resultados das atividades de teste desenvolvidas e fornece uma avaliação baseada nestes resultados. Deve referir todos os itens testados e toda documentação relativa ao teste.

2.6 Quem deve testar e quando?

Vamos discutir, neste momento, em que ponto do processo de desenvolvimento as atividades de teste devem ser realizadas e também qual o papel de cada uma das pessoas envolvidas nessas atividades. Nesse processo de teste, os documentos relativos ao teste são gerados (os quais foram apresentados na seção anterior) e a quantificação deles para cada etapa é apresentada nas tabelas 1, 2 e 3, observando que K, M, N e P são números inteiros arbitrários:

Tabela 1:Teste de UI

(1) Plano de Teste	por UI
(P) Projetos de Teste	por UI
(M) Procedimentos de Teste	por Projeto de Teste
(N) Casos-de-Teste	por UI
(M) Registros de Teste	por Projeto de Teste
(K) Relatórios de Problemas de Teste	por UI
(1) Resumo de Teste	por UI

Tabela 2:Teste de Integração

(1) Plano de Teste	por subsistema

Tabela 2:Teste de Integração

(P) Projetos de Teste	por subsistema
(M) Procedimentos de Teste	por Projeto de Teste
(N) Casos-de-Teste	por subsistema
(M) Registros de Teste	por Projeto de Teste
(K) Relatórios de Problemas de Teste	por subsistema
(1) Resumo de Teste	por subsistema

Tabela 3:Teste de Sistema

(1) Plano de Teste	por sistema
(P) Projetos de Teste	por sistema
(M) Procedimentos de Teste	por Projeto de Teste
(N) Casos-de-Teste	por sistema
(M) Registros de Teste	por Projeto de Teste
(K) Relatórios de Problemas de Teste	por sistema
(1) Resumo de Teste	por sistema

Durante o processo de teste, uma série de pessoas participa de diversas atividades. Vamos mostrar agora quem são estes personagens.

Tabela 4: Recursos humanos para teste

Gerente de Projeto:	é o agente responsável pelos assuntos administrativos relativos ao desenvolvimento do projeto.
Gerente Técnico de Projeto:	é o agente responsável pela condução dos assuntos técnicos relativos ao desenvolvimento do projeto.
Equipe de Teste:	é o grupo formado por elementos responsáveis pela condução dos testes (integração e sistema) e pela garantia de qualidade do projeto.

Analista:	é o agente responsável pelo desenvolvimento ou manutenção dos módulos automatizados do sistema de informação.
Programador:	é o agente responsável pela construção e teste das unidades e suas adequações ao projeto físico.

Por meio das tabelas abaixo, serão indicados os documentos que são gerados durante o processo de teste, quem os gera, quem os revisa, que documentos são necessários para sua consecução e em que ponto do ciclo de desenvolvimento do projeto eles devem ser gerados. As tabelas mostram estas relações para os testes de UI, para os testes de integração e para os testes de sistema. A lista de documentos (e suas siglas) referidos como de entrada é dada abaixo:

- Padrões Adotados
- Especificação de Objetivos e Requisitos (OR)
- Descrição Funcional (DF)
- Projeto de Interface Homem-Máquina (PI)
- Manual de Usuário (MU)
- Especificação de Definição das Interfaces (EC)
- Especificação de Características da UI (CA)
- Mapeamento Funcional Físico (MF)
- Arquivo Fonte (AT)
- Arquivo Executável (AE)
- Plano de Teste (PN)
- Projeto de Teste (PJ)
- Caso-Teste (CS)
- Procedimento de Teste (PR)

- Registro de Teste (RG)
- Relatório de Problemas de Teste (RT)
- Relatório de Resumo de Teste (RE)

.

Tabela 5: Documentos para teste de UI

TESTE DE UI				
Documentos Gerados	**Executores**	**Revisores**	**Documentos de Entrada**	**Fase da MUSA**
Plano de Teste	Gerente Técnico de Projeto	Equipe de Teste	CA, EC	Projeto Físico
Projeto de Teste	Gerente Técnico de Projeto	Equipe de Teste	PN, CA, EC	Projeto Físico
Caso-de-Teste	Equipe de Teste	Equipe de Teste	PJ, CA, EC	Projeto Físico
Procedimento de Teste	Equipe de Teste	Equipe de Teste	PJ, CS	Implementação
Registro de Teste	Equipe de Teste	Equipe de Teste	PN, PJ, CS, PR, AE	Implementação
Relatório de Problema de Teste	Equipe de Teste	Equipe de Teste	PN, PJ, CS, PR, AE	Implementação
Relatório de Resumo de Teste	Equipe de Teste	Equipe de Teste	RG, RT, PN	Implementação

Tabela 6: Documentos para teste de integração

TESTE DE INTEGRAÇÃO				
Documentos Gerados	**Executores**	**Revisores**	**Documentos de Entrada**	**Fase da MUSA**
Plano de Teste	Gerente Técnico de Projeto	Equipe de Teste	DF, EC	Projeto Lógico
Projeto de Teste	Gerente Técnico de Projeto	Equipe de Teste	PN, DF, EC	Projeto Físico
Caso-de-Teste	Equipe de Teste	Equipe de Teste	PN, PJ, DF, EC	Projeto Físico
Procedimento de Teste	Equipe de Teste	Equipe de Teste	PJ, CS	Implementação

Tabela 6: Documentos para teste de integração

TESTE DE INTEGRAÇÃO				
Registro de Teste	Equipe de Teste	Equipe de Teste	PN, PJ, CS, PR, AE	Implementação
Relatório de Problema de Teste	Equipe de Teste	Equipe de Teste	PN, PJ, CS, PR, AE	Implementação
Relatório de Resumo de Teste	Equipe de Teste	Equipe de Teste	RG, RT, PN	Implementação

Tabela 7: Documentos para teste de sistema

TESTE DE SISTEMA				
Documentos Gerados	**Executores**	**Revisores**	**Documentos de Entrada**	**Fase da MUSA**
Plano de Teste	Gerente Técnico de Projeto	Equipe de Teste	OR, PI	Especificação Requisitos
Projeto de Teste	Gerente Técnico de Projeto	Equipe de Teste	PN, OR, PI	Especificação Requisitos
Caso-de-Teste	Equipe de Teste	Equipe de Teste	PN, PJ, OR, PI, DF	Projeto Lógico
Procedimento de Teste	Equipe de Teste	Equipe de Teste	PJ, CS	Implementação
Registro de Teste	Equipe de Teste	Equipe de Teste	PN, PJ, CS, PR, AE	Implementação
Relatório de Problema de Teste	Equipe de Teste	Equipe de Teste	PN, PJ, CS, PR, AE	Implementação
Relatório de Resumo de Teste	Equipe de Teste	Equipe de Teste	RG, RT, PN	Implementação

2.7 Ambiente de Software

O ambiente de desenvolvimento é composto por ferramentas CASE - *Computer Aided Software Engineering* - adquiridas de diversos fornecedores e adaptadas para atender às necessidades do CPqD e aos padrões da MUSA.

A ferramenta base é a de apoio à análise e projeto. Ela é uma ferramenta que apóia, por meio de editores gráficos, a construção de modelos de entidades e relacionamentos (notações de Chen e

Bachman com extensões); a construção de diagramas de fluxo de dados na análise estruturada (notações de DeMarco/Yourdon[72] e Gane/Sarson[26]); a construção de diagramas de estruturas de dados segundo as convenções propostas por Jackson[41]; e a construção de diagramas de estrutura segundo Yourdon/Constantine[73]. Essa ferramenta também possui módulos para geração automática de código (C e C++) e para realização de engenharia reversa (de código C para o diagrama de estrutura). Também estão disponíveis para os projetistas todo o conjunto de ferramentas do projeto GNU, como compiladores, ligadores, editores de texto etc.

O desenvolvimento e a depuração de código na linguagem C são auxiliados por ferramentas que contribuem para garantir a consistência entre o código e os diagramas de especificação e projeto. Qualquer alteração efetuada no código pode ser automaticamente disseminada para os diagramas do nível físico e vice-versa.

A documentação dos sistemas é feita segundo os padrões dos meta-documentos estabelecidos pela MUSA e que são baseados nas recomendações do IEEE[34]. As ferramentas de apoio utilizadas permitem a combinação e manutenção de texto, gráficos, diagramas e tabelas em um mesmo documento.

O projeto da interface de usuário é auxiliado por um Sistema de Gerência de Interface de Usuário (*UIMS*) que segue o modelo SEEHEIM de particionamento de código, que foi estabelecido pela X/Open Technology. Esse modelo prevê a descrição da interface de usuário em três camadas, uma camada de apresentação responsável pela interface com o usuário final onde são descritos os objetos de interação visual, uma camada de diálogo onde se descrevem os elementos estruturais do diálogo e o comportamento dos objetos de interação e uma camada de aplicação responsável pela interface com a aplicação controlada pela interface. O nível de apresentação é elaborado de acordo com as recomendações do padrão OSF/Motif e com o auxílio de um editor de *layout* de telas. O nível de diálogo é programado numa linguagem interpretada que possibilita a prototipagem rápida. A construção de protótipos é extremamente importante uma vez que todos os usuários também estão envolvidos na tarefa de especificação. O código na linguagem C é automaticamente gerado tanto para os *layouts* das telas quanto para os programas de diálogo.

2.8 Considerações finais

Neste capítulo, foi caracterizado que o software está presente nas diversas famílias de sistemas de telecomunicações e foi apresentada uma dessas famílias de sistemas, que é a família dos sistemas de suporte à operação. Foram apresentados o ambiente e a metodologia de desenvolvimento usadas no CPqD para o desenvolvimento de alguns desses sistemas. Para a metodologia foi apresentada sua estruturação em etapas e como o processo de teste está inserido nessa metodologia. O processo de teste também é dividido em etapas (planejamento, projeto, implementação e

execução) e é apoiado por uma série de documentos. Particularmente, foi ressaltado que atualmente o teste de unidade é uma atividade informal e que vários autores concordam que a atividade de teste deve ser conduzida ao longo da construção do software. Com isso a proposta de criar um ambiente que torne o teste de unidade uma atividade sistemática, caminha no sentido proposto por esses autores e pode se constituir numa contribuição para a melhoria da própria metodologia. Ao final foi apresentado o ambiente de desenvolvimento utilizado. No próximo capítulo serão tratados o teste e a gestão de configuração de software.

Capítulo 3

Teste e Gestão de Configuração de Software

3.1 Introdução

O presente capítulo tem por objetivo tratar dos aspectos teóricos dos dois assuntos básicos do presente trabalho: o teste e a gestão de configuração de software. Ele dará uma descrição, em linhas gerais, das atividades de teste com vistas a validar e verificar um produto de software que está sendo desenvolvido. Aqui as seguintes perguntas serão respondidas:

- O que testar?
- Como testar?
- Onde testar?

Do ponto de vista da gestão de configuração, será apresentado o processo de gestão, as funções necessárias, os recursos humanos, o conceito de configuração-base (*baseline*), respondendo as seguintes perguntas:

- Por que gerenciar a configuração?
- O que controlar?
- Como controlar?
- Quem deve gerenciar?
- Quando controlar

3.2 O teste de software

3.2.1 O que testar?

A atividade de teste dentro de um projeto de software pode ser classificada em cinco níveis de atuação, dependendo do objeto que está sendo testado:

1. Teste de Unidade

2. Teste de Unidade de Implementação (UI)

3. Teste de Integração

4. Teste de Sistema

5. Teste de Aceitação

Deve-se lembrar que a nomenclatura aqui adotada difere um pouco da usual, pois ela está de acordo com a utilizada pela MUSA[52], explicada no capítulo anterior. Esta diferença ficará mais clara a seguir.

3.2.1.1 Teste de unidade

O teste de unidade é o primeiro e mais básico teste que o software de um projeto pode sofrer. Deve-se lembrar que unidade, aqui é entendida como sendo um componente indivisível de código sob a responsabilidade de apenas um programador.

Em geral, esse teste é dividido em duas fases. A primeira se concentra em depurar a unidade em seu contexto léxico-sintático (por exemplo, por meio do uso de compiladores); nesse ponto, o uso de listas de verificação (*check-list*) e inspeção visual do código também traz bons resultados. Já a segunda tem por objetivo verificar a lógica da unidade. Nessa fase, a proposta é exercitar as interfaces, os comandos iterativos, os comandos condicionais, a manipulação das estruturas de dados e verificar o fluxo de controle.

A importância do teste de unidade, segundo Humphrey[31], se dá pelo fato de que, quando os módulos não são unitariamente testados, o teste de integração pode sofrer grandes atrasos devido às paralisações para correções de módulos individuais. Além do fato de que o custo de alteração de um software é tanto mais baixo quanto mais cedo um defeito for encontrado.

3.2.1.2 Teste de Unidade de Implementação

À medida em que as unidades vão sendo testadas, podemos ir iterativamente construindo o software completo das UI pela adição dessas unidades. O teste de UI serve para exercitar a UI de acordo com o especificado na documentação do projeto. Pode-se utilizar uma abordagem *top-down* ou *bottom-up* para construção e teste do software; no entanto, na abordagem *top-down* temos a vantagem de que a maioria das interfaces são definidas num nível mais alto da hierarquia e, portanto, verificadas mais cedo. Com isso, uma vez verificadas as interfaces, a adição das diferentes unidades já testadas fica facilitada.

Sua proposta é verificar que:

- as várias unidades se interfaceiam de maneira correta para formar as UIs como descrito no documento CA,
- as UIs foram corretamente codificadas, de tal modo que toda a sua funcionalidade opere de maneira correta e todos os caminhos lógicos fluam como o projetado,
- todos os estados e todas as transições de estado das UIs tenham sido exercitadas e
- haja uma progressão ordenada em direção à implementação das características fins do software, as quais serão verificadas pelo Teste de Sistema.

Este teste é constituído de dois subtestes: o Teste de Integração de Unidades e o Teste de Verificação Funcional. O primeiro deve verificar a correção das interfaces das UIs, observando se cada UI pode chamar ou ser chamada por outras UIs que possuam interface com ela, e também verificar se os parâmetros são passados corretamente. Os objetivos principais desse subteste são:

1. verificar a compatibilidade entre unidades que interagem entre si e
2. verificar a compatibilidade das unidades com suas interfaces externas.

O Teste de Verificação Funcional deve examinar a coleção de unidades já implementadas e testá-las para observar se elas operam como um módulo integrado e se executam as funções que lhes foram atribuídas no documento CA.

3.2.1.3 Teste de Integração

Este é um teste de integração funcional. A idéia é reunir as UIs do Sistema Físico (que foi especificado na Fase de Projeto Físico) em módulos e testá-los com relação ao descrito no documento de especificação de requisitos funcionais.

3.2.1.4 Teste de Sistema

Este é o teste executado com o software completo antes que ele seja entregue ao cliente. Ele deve garantir que os requisitos do cliente (estabelecidos na Especificação de Requisitos de Software) foram implementados. Nesse teste, as interfaces externas são verificadas e demonstra-se que o software satisfaz os requisitos funcionais e de desempenho especificados.

3.2.1.5 Teste de Aceitação

Segundo o padrão ISO 9000-3[37] o Teste de Aceitação serve para o cliente "julgar se o produto é, ou não, aceitável de acordo com os critérios préviamente acordados, e da maneira especificada em contrato". A proposta deste teste é demonstrar que o produto executa como o esperado no ambiente do usuário final e com base de dados real. A priori, é idêntico ao teste de sistema, menos quanto ao ambiente onde é realizado.

3.2.2 Como testar?

Se olharmos o programa como sendo uma função que transforma elementos de um dado conjunto domínio, para um outro conjunto imagem, podemos dizer que cabe ao teste: escolher entradas pertencentes ao conjunto domínio, determinar as saídas esperadas, executar o programa e comparar as saídas esperadas com as obtidas. É importante lembrar, também, que Myers[51] deixa bem claro o objetivo do teste como sendo o de revelar a presença de erros no software e que um bom teste é aquele que efetivamente consegue revelar os erros do software em teste.

Um produto software pode ser testado por meio de duas formas[57]:

1. conhecendo-se as funções que foram especificadas para o software executar, o teste pode ser conduzido para demonstrar a operacionalidade das funções, ou seja, esse teste é baseado na especificação do software (*specification-based testing*) e

2. conhecendo-se a estrutura interna do software, o teste pode ser conduzido para verificar se todos os componentes internos, quando exercitados, operam de maneira adequada, ou seja, o teste é baseado nas características da implementação (*program-based testing*).

Esses dois modos de se testar um software são implementados pelas três técnicas de teste mais conhecidas: a funcional, a estrutural e a baseada em erros. Deve-se observar que essas técnicas de teste não são mutuamente exclusivas, mas, sim, complementares e, portanto, devem ser usadas em conjunto de forma a propiciar uma atividade de teste mais eficiente e eficaz[57].

Argumenta-se que entre as três técnicas de teste mais conhecidas, a técnica estrutural tem a mais alta capacidade de detectar erros. Segundo Humphrey[31], um estudo efetuado por Thayer e Lipow mostrou que na média a técnica estrutural é capaz de detectar 72,9% de todos erros encontrados por usuários de um grande sistema. No entanto, essa técnica possui algumas limitações e desvantagens como a determinação de caminhos e associações não executáveis e o problema de caminhos ausentes (quando uma funcionalidade deixa de ser implementada no programa, não

existe um caminho que corresponda àquela funcionalidade e, portanto, nenhum caso de teste será requerido para exercitá-la)[48].

A técnica funcional, por sua vez, é até mais intuitiva, pois se baseia em "o que o software deve fazer". Mas a dependência dessa técnica na especificação do programa gera um problema. Uma vez que a maioria das especificações são feitas de modo informal, os requisitos de teste dela derivados também acabam sendo informais e imprecisos[48].

Daí a afirmação de Pressman[57] sobre a complementaridade das técnicas e a afirmação de Beizer[4] de que nenhuma pode ser considerada superior à outra; a utilidade de uma técnica depende da natureza do objeto em teste, da natureza dos erros encontrados no objeto em teste e até do estágio de conhecimento do testador. Ele também afirma que é natural que nos estágios iniciais do desenvolvimento (teste de unidade) o teste estrutural seja mais utilizado e que nos estágios finais (teste de sistema) o teste funcional seja mais utilizado.

3.2.2.1 Teste funcional

O teste funcional (ou caixa-preta) é realizado olhando-se o software apenas através de suas interfaces, portanto, testando sua funcionalidade. Os erros encontrados através dessa técnica pertencem às seguintes categorias:

- funções incorretas,
- erros de interface,
- erros na estrutura de dados ou no acesso a dados externos,
- erros de desempenho e
- erros de iniciação ou finalização.

Segundo Myers[51] o uso dessa técnica:

- reduz o número de casos de teste necessários para se executar um teste confiável,
- produz casos de teste que indicam a presença ou ausência de classes de erros e não apenas erros associados ao teste em questão.

Um problema que pode ser levantado com relação ao teste funcional está no próprio processo de desenvolvimento de software o qual, na maioria das organizações, ainda está muito incipiente. O

relatório do MCT/SEPIN[59] sobre o setor de software no Brasil aponta que apenas 1,8% das empresas conhece e usa o modelo CMM e que apenas 14% administra seus requisitos. Isso nos leva a crer que a maioria das empresas não tem uma abordagem adequada com relação à atividade mais importante para o teste baseado em especificação, ou seja, a especificação de requisitos que, em geral, é feita de forma bastante informal e prejudicial à atividade de derivação de dados de teste. Alguns dos principais critérios associados a essa técnica, apresentados em[57] são:

Particionamento de Equivalência

Como o teste exaustivo é impossível, esta técnica prevê que o ideal seria selecionar um subconjunto da entrada que possua uma grande probabilidade de encontrar o maior número de erros possíveis no programa. Um caso de teste bem escolhido deve atender a duas propriedades: deve garantir que cada caso de teste contenha tantas condições de entrada quanto for possível para minimizar o número de casos de teste necessários; e também que o domínio da entrada seja particionado em classes (válidas e inválidas), de tal maneira que, com alguma certeza, possamos garantir que um teste executado com um elemento de alguma classe tenha resultado igual ao que teria para qualquer elemento dessa mesma classe, ou seja, todos devem passar ou todos devem falhar.

Análise de Valor Limite

Nesta técnica, a idéia é verificar as condições que ficam exatamente acima e abaixo dos limites do domínio de entrada. A análise de limite difere do particionamento de equivalência em dois aspectos:

1. Ao invés de selecionar "qualquer" elemento em uma classe de equivalência, ela requer que um ou mais elementos sejam selecionados, de tal modo que os limites dessa classe sejam objetos de teste; e

2. Além de considerar as condições de entrada, os casos de teste devem ser derivados considerando-se também o conjunto de saída, ou seja, as classes de equivalência de saída.

Grafo de Causa-Efeito

As duas técnicas já apresentadas não exploram a combinação de elementos da entrada. A combinação destes elementos não é uma tarefa fácil porque o número de combinações pode ser inaceitável, do ponto de vista de aplicação prática. Essa outra técnica ajuda a selecionar, de modo sistemático, um conjunto de casos de teste levando em conta esta possibilidade de combinação. O grafo de Causa-Efeito é uma linguagem formal, para a qual uma especificação em linguagem natural é traduzida, sendo que o grafo resultante representa um circuito lógico digital. É necessário ter algum conhecimento de lógica Booleana (isto é, conhecimento dos operadores E, OU e NÃO). A idéia principal é relacionar condições com as respectivas ações, ou seja, as entradas (causas) são relacionadas com as saídas (ações) de um módulo, um grafo é contruído a partir dessas informações, este grafo é transformado numa tabela de decisão e então as regras desta tabela são usa-

das para gerar os casos de teste. Devido a este formalismo, essa técnica não tem seu uso difundido. Existe, até mesmo, uma crítica sobre o fato de o testador ser obrigado a transfomar uma representação complexa (o programa) em outra tão complexa quanto (o grafo)[4].

3.2.2.2 Teste estrutural

O teste estrutural (ou caixa-branca) tem o aspecto de exame procedimental do software, ou seja, um exame do comportamento do software baseado em sua estrutura interna. Por meio deste teste, dependendo dos critérios associados adotados, pode-se, por exemplo:

- garantir que todos os comandos de um dado programa tenham sido executados pelo menos uma vez,
- exercitar todas as decisões lógicas (verdadeiro/falso),
- executar todos os comandos iterativos nos seus limites e em sua faixa operacional e
- exercitar a estrutura de dados interna para assegurar sua validade.

Em geral, a maioria dos critérios dessa técnica utiliza uma representação de programa conhecida como **grafo de fluxo de controle** ou **grafo de programa**. A representação de um programa *P* como um grafo de fluxo de controle *(G = (N, E, s))* consiste em estabelecer uma correspondência entre nós e blocos e em indicar possíveis fluxos de controle entre blocos através dos arcos. Um grafo de fluxo de controle é, portanto, um grafo orientado, com um único nó de entrada $s \in N$ e um único nó de saída, no qual cada vértice representa um bloco indivisível de comandos e cada aresta representa um possível desvio de um bloco para outro. Cada bloco tem as seguintes características: 1) uma vez que o primeiro comando do bloco é executado, todos os demais são executados seqüencialmente; 2) não existe desvio de execução para nenhum comando dentro do bloco. A partir do grafo de programa, podem ser escolhidos os componentes que devem ser executados, caracterizando, assim, o teste estrutural.

Seja um grafo de fluxo de controle *G = (N, E, s)* onde *N* representa o conjunto de nós, *E*, o conjunto de arcos e *s*, o nó de entrada. Um **caminho** é uma seqüência finita de nós $(n_1, n_2, ..., n_k)$, $k >= 2$, tal que exista um arco de n_i para n_{i+1} para $i = 1,2,...,k-1$. Um caminho é um **caminho simples** se todos os nós que compõem esse caminho, exceto, possivelmente, o primeiro e o último, são distintos; se todos os nós são distintos diz-se que esse caminho é um **caminho livre de laço**. Um **caminho completo** é um caminho onde o primeiro nó é o nó de entrada e o último nó é o nó de saída do grafo *G*. Seja *IN(x)* e *OUT(x)* o número de arcos que entram e que saem do nó *x*,

respectivamente. Se $IN(x) = 0$ x é um nó de entrada, e se $OUT(x) = 0$, x é um nó de saída. Os principais critérios associados a essa técnica são dados a seguir:

Fluxo de Controle

Os critérios desta família estão associados às estruturas de controle do fluxo do programa. Os mais conhecidos são Todos-os-nós, onde a exigência é de que a execução do programa passe, pelo menos uma vez, em cada nó do grafo do programa (também conhecido por cobertura de comando); Todos-os-arcos, onde a exigência é de que cada aresta do grafo do programa seja exercitada pelo menos uma vez, testando, assim, cada desvio de fluxo de controle; e Todos-os-caminhos, onde a exigência é de que todos os caminhos possíveis sejam executados[57]. Um outro critério, mais poderoso que Todos-os-nós e Todos-os-arcos, é a cobertura de decisão, em que se exige que todas as estruturas de decisão sejam testadas para todas as suas possibilidades de saída, ou seja, no caso de um comando de decisão tipo IF, devem ser testadas as saídas positiva (*then*) e negativa (*else*). Myers mostra alguns pontos fracos nestes critérios, como erros de lógica de decisão que não seriam acusados mesmo com casos de teste adequados ao critério. O critério cobertura de condição exige que cada condição, numa decisão, assuma todos os valores possíveis pelo menos uma vez. Mais detalhes sobre o uso combinado desses critérios, o qual pode gerar critérios mais poderosos, como critério de cobertura de decisão e condição, podem ser encontrados em[51].

Fluxo de Dados

Os critérios baseados em fluxo de dados (*Data Flow*) utilizam informações sobre o fluxo de dados do programa para conseguir levantar os requisitos necessários ao teste, os quais são obtidos a partir da interação que existe entre as definições de variáveis e o uso dessas definições[57]. De modo simplificado, podemos dizer que a definição de uma variável se dá quando um valor é atribuído a ela e um uso dessa variável se dá quando esse valor é utilizado [48].

Exemplos de critérios dessa família são os critérios Todas-as-definições, Todos-os-usos e os Potenciais-usos[47]. O critério Todas-as-definições requer que pelo menos um uso de cada definição de variável seja exercitado. O critério Todos-os-usos exige que todas as associações entre as definições de uma variável e seus usos sejam exercitadas, ao menos, uma vez, por um caminho livre de definição (ou seja, um caminho no qual essa variável não seja redefinida). Os critérios Potenciais-usos são uma extensão do conceito de definição-uso, introduzindo a idéia de que, mesmo não havendo um uso explícito de uma variável, a existência de um caminho livre de definição entre um nó onde há uma definição e um outro nó, caracteriza uma potencial-associação entre a definição e o potencial-uso.

Teste de Laço

Esta técnica foi proposta por Boris Beizer[3] e tem por objetivo único validar os comandos de repetição (*loops*). Ele identificou 4 classes diferentes de comandos de repetição:

- Comando simples:

Veja abaixo os testes necessários para testar um laço com limite de execução "n".

1. Saia da repetição (force a saída).
2. Passe só uma vez pelo laço.
3. Passe duas vezes pelo laço.
4. Passe "m" vezes pelo laço (sendo m<n).
5. Passe "n-1", "n", "n+1" pelo laço.

- Comandos aninhados:

1. Concentre-se primeiro no laço mais interno, testando-o como um laço simples e mantendo os outros laços em seus valores de controle mínimos.
2. Mantenha o laço testado em um outro valor típico e teste agora o próximo mais externo, mantendo os outros em seus valores de controle mínimos.
3. Continue até testar todos os laços.

- Comandos concatenados:

Neste caso, se os comandos forem independentes, eles podem ser testados como laços simples, mas, caso o contador de um deles seja usado como valor inicial do outro, então, eles devem ser testados como laços aninhados.

- Comandos não estruturados:

Neste caso, o comando deve ser reprojetado de modo a refletir uma das três classes discutidas anteriormente.

3.2.2.3 Teste baseado em erros

A técnica de teste baseada em erros utiliza informações sobre os tipos de erros mais freqüentes no processo de desenvolvimento de software, para derivar os requisitos de teste. A ênfase da técnica

está nos erros que o programador ou projetista pode cometer durante o desenvolvimento e nas abordagens que podem ser usadas para detectar a sua ocorrência. **Semeadura de Erros** (*Error Seeding*)[13] e **Análise de Mutantes-AM** (*Mutation Analysis*)[22] são critérios típicos que se concentram em erros. A seguir, é apresentado o critério **AM**.

O Critério Análise de Mutantes-AM

Os fundamentos dessa técnica estão baseados em duas hipóteses, a **hipótese do programador competente** (*competent programmer hypothesis*) e o **efeito de acoplamento** (*coupling effect*)[22]. Na primeira, pressupõe-se que programadores experientes escrevam programas corretos ou muito próximos do correto. Dessa forma, pode-se afirmar que os erros são introduzidos nos programas através de pequenos desvios sintáticos que, embora não causem erros sintáticos, alteram a semântica do programa e, conseqüentemente, conduzem o programa a um comportamento incorreto. Para revelar tais erros, a **AM** identifica os desvios sintáticos mais comuns e, através da aplicação de pequenas transformações sobre o programa em teste, encoraja o testador a construir casos de testes que mostrem que tais transformações levam a um programa incorreto[48]. A segunda hipótese pressupõe que erros complexos estão relacionados a erros simples e alguns estudos empíricos já confirmaram esta hipótese[48], mostrando que conjuntos de casos de teste capazes de revelar erros simples são também capazes de revelar erros complexos.

Partindo-se das duas hipóteses apresentadas, a princípio, o testador deve fornecer um programa P a ser testado e um conjunto de casos de teste T cuja adequação se deseja avaliar. O programa é executado com T e, se apresentar resultados incorretos, então, um erro foi encontrado e o teste termina. Caso contrário, o programa ainda pode conter erros que o conjunto T não conseguiu revelar. O programa P sofre, então, pequenas alterações, dando origem aos programas P_1, P_2,..., P_n denominados **mutantes** de P, os quais diferem de P apenas pela ocorrência de erros simples. Para modelar esses desvios sintáticos mais comuns, **operadores de mutação** (*mutant operators*) são aplicados a um programa P, transformando-o em programas similares: mutantes de P (por exemplo, trocando o nome de uma variável ou trocando um operador lógico).

A seguir, os mutantes são executados com o mesmo conjunto de casos de teste T. O objetivo é obter casos de teste que resultem apenas em mutantes mortos (um mutante é considerado morto quando, para algum caso de teste, o resultado da execução do mutante e o do programa original diferem entre si) e mutantes equivalentes (um mutante é considerado equivalente quando o mutante e o programa original apresentam sempre o mesmo resultado, para qualquer $d \in D$); neste caso, tem-se um conjunto de casos de teste T adequado ao programa P em teste, no sentido de que, ou P está correto, ou possui erros pouco prováveis de ocorrerem. É preciso ressaltar que,

em geral, a equivalência entre programas é uma questão indecidível e requer a intervenção do testador, que atua como um oráculo decidindo sobre a equivalência.

Um ponto importante, que deve ser destacado sobre a **AM**, é o fato dela fornecer uma medida objetiva do nível de confiança da adequação dos casos de teste analisados, através da definição de um **escore de mutação** (*mutation score*), que relaciona o número de mutantes mortos com o número de mutantes gerados[48]. O escore de mutação é calculado da seguinte forma:

$$ms(P, T) = (DM(P, T))/((M(P)) - EM(P))$$

onde:

DM(P,T): número de mutantes mortos pelos casos de teste em T.

M(P): número total de mutantes gerados.

EM(P): número de mutantes gerados equivalentes a P.

O escore de mutação varia no intervalo entre 0 e 1 sendo que, quanto maior o escore mais adequado é o conjunto de casos de teste para o programa testado. Percebe-se, com essa fórmula, que apenas *DM(P,T)* depende do conjunto de casos de teste utilizado e que *EM(P)* é obtido à medida que o testador, manualmente ou com o apoio de heurísticas, decide que determinado mutante vivo é equivalente[62].

No entanto, um dos maiores problemas para a aplicação do critério **AM** está relacionado ao seu alto custo, uma vez que o número de mutantes gerados, mesmo para pequenos programas, pode ser muito grande, exigindo um tempo de execução muito extenso. Várias estratégias têm sido propostas para fazer com que a **AM** possa ser utilizada de modo mais eficiente, dentro de limites economicamente viáveis. Além disso, critérios alternativos derivados da AM também foram criados com o intuito de reduzir o custo a ela associado: **Mutação Aleatória** (*Randomly Selected X% Mutation*), **Mutação Restrita** (*Constrained Mutation*) e **Mutação Seletiva** (*Selective Mutation*). Tais critérios procuram selecionar apenas um subconjunto do total de mutantes gerados, reduzindo o custo associado, mas com a expectativa de não se reduzir a eficácia do critério[48].

3.2.3 Ferramentas para teste de software

A crescente complexidade dos sistemas de software faz com que qualquer iniciativa no sentido da sistematização da atividade de teste se transforme numa tarefa trabalhosa e propensa a erros.

A falta de uma ferramenta de teste que automatize a aplicação efetiva de um critério de teste, faz com que sua aplicação seja restrita apenas a programas muito simples[30]. E mais, Beizer[4] acrescenta que testes manuais são anacrônicos, apresenta uma estatística de erro de testes manuais, explica que testes manuais levam a se obter uma falsa confiança no software testado e que se os cálculos fossem bem feitos, as organizações ficariam chocadas em saber o custo real da atividade de teste manual.

As ferramentas são importantes para reduzir a intervenção humana na atividade de teste, aumentando a produtividade e influenciando na confiabilidade do software testado. Elas também fornecem suporte para o teste de regressão, já que os casos de teste utilizados numa dada sessão de teste podem ser, facilmente, obtidos para a revalidação de um software após uma alteração.

Podemos encontrar ferramentas para apoio aos critérios de fluxo de controle e de dados, como Poke-tool[16][17] e STW/COV[61]; para apoio aos critério análise de mutantes, como Proteum[20] e Proteum/IM[21]; ferramentas para *capture/playback* (que gravam tudo que acontece durante o teste - como movimentos de mouse, valores inseridos, teclas usadas, elementos de interface gráfica acionados etc. - e depois podem executar outra vez esse mesmo teste, mas para uma nova versão do programa executável em teste), como STW/CAPBAK[61] ou XRunner[50]; bem como ferramentas para geração de dados de teste, como a T[32] ou o protótipo desenvolvido na UNICAMP[14].

3.2.4 Onde testar?

Esta discussão pode ser considerada filosófica, pois sua resposta depende da metodologia aplicada. Quanto aos três primeiros níveis de teste (unidade, integração e sistema), não há dúvidas de que eles devem ser executados no ambiente de desenvolvimento (sendo o Teste de Sistema executado em ambiente que simula o ambiente do usuário final). Já o Teste de Aceitação deve ser executado no ambiente alvo, em condições reais de operação. No entanto, nada impede que em contrato seja estabelecido que ele também seja executado no ambiente de desenvolvimento, mas sob supervisão do cliente e utilizando dados reais.

3.3 Gestão de configuração

Para aqueles que não acreditam em disciplina, a seguinte frase cabe como uma luva:

Se você não acredita em disciplina, observe o movimento de um carro sem freio.

Há muito que a produção de software teve de deixar de lado sua característica de produção doméstica para adotar um caráter mais sério. Mesmo assim, acreditava-se que a disciplina estaria

ligada somente à maneira de escrever programas. Não há dúvidas de que isto muito ajudou, mas cedo se descobriu que, para projetos grandes, o gerenciamento do processo como um todo era indispensável. Principalmente, quando a evolução de um projeto (forma e conteúdo) tornava-se desconhecida do próprio gerente do projeto. Cenas como a de um projetista testando um módulo que havia sido modificado por outro projetista, sem ter sido avisado, são comuns em ambientes como este.

Bersoff[7] garante que a Gestão de Configuração de Software (GCS) nasceu como resposta aos erros cometidos durante os anos 70, quando computadores começaram a ser utilizados de maneira geral para a solução dos mais diversos e complexos problemas para os quais o gerenciamento tradicional mostrou-se inadequado. Principalmente, quando se descobriu que a complexidade de gerenciamento de um sistema não variava de maneira linear com o número de linhas de código, mas de maneira exponencial. Como observa Babich[2], ela é uma disciplina útil para a gerência do projeto, pois lhe permite executar tarefas complexas de uma maneira mais organizada, o que proporciona aumento de produtividade e redução de erros.

3.3.1 O que é GCS?

No processo de produção de software, um objeto em desenvolvimento não tem um comportamento estático, ele evolui com o tempo. Chama-se de versão ao estado particular de um objeto e, por configuração, entende-se a relação entre as versões de um objeto composto, ou seja, configuração é uma instância do sistema composta da união de uma versão específica de cada objeto componente[65].

Gestão de Configuração de Software não fornece um método de projeto, nem um modelo de ciclo de vida, nem, tampouco, define como a qualidade dos itens deve ser julgada; ela fornece um fundamento sólido para todas as outras atividades de engenharia de software[15]. Podemos defini-la como uma disciplina para gerenciamento efetivo da produção de software. Cabe a ela identificar a configuração de um sistema em relação ao tempo com a finalidade de, sistematicamente, controlar as mudanças desta configuração, manter sua integridade e, desta forma, possibilitar seu rastreamento através do ciclo de vida do sistema[6].

As funções da gestão de configuração são: identificação de configuração (que itens constituem uma configuração), controle de configuração (que passos no processo de alteração afetam uma configuração), auditoria de configuração (quais são as diferenças entre as versões) e contabilização da situação de configuração (que modificações foram feitas por determinado programador)[6][12].

Uma característica importante de um gerenciador é o modo de armazenamento dos objetos, já que isto implica em uso de espaço. Alguns modos propostos são: orientação por arquivos, orientação por páginas[45], orientação por registros[43] e orientação por deltas (utilizado pelo `SCCS`[9] e pelo `RCS`[63]). Além do modo, temos também de definir um sistema de armazenamento. O `SCCS` e o `RCS` são gerenciadores implementados sobre o sistema operacional UNIX (neles uma versão é reconstruída a partir de seus arquivos delta, os quais armazenam as diferenças entre as várias versões do mesmo item); o `DSEE`[44] e o `Gypsy`[18] possuem o controle de versões integrado ao sistema operacional (e uma versão pode ser diretamente recuperada pelo sistema operacional), já o `Adele`[5], o `CCC/Manager`[60] e o `MVC`[66] são gerenciadores onde o núcleo do sistema é baseado em banco de dados.

A integridade dos dados é garantida pelo controle de acesso. Gerenciadores como o RCS utilizam, além da proteção mínima provida pelo sistema operacional, uma lista de acesso, na qual cada objeto detém uma lista de usuários que têm direitos sobre ele. O CCC, por exemplo, divide os usuários em classes e define o acesso e as operações sobre os objetos segundo essas classes.

3.3.2 Por Que Gerenciar a Configuração?

O pessoal ligado a *off-road*, os conhecidos jipeiros, possuem um lema que diz:

Se você não sabe para onde vai, qualquer caminho serve.

Para entendermos porque gerenciar a configuração de um sistema, talvez devêssemos começar por entender os objetivos do processo de desenvolvimento de software, ou seja, para onde ele nos leva[7]. O resultado deste processo é um produto, que é fruto da transformação de uma idéia, que, por sua vez, traduz as necessidades de uma pessoa (ou um grupo de pessoas). Portanto, podemos dizer que o objetivo principal do processo de desenvolvimento de software é (ou deveria ser) a construção de produtos que satisfaçam as reais necessidades dessa pessoa (ou grupo) a quem o software é destinado. Essa realização de objetivos é conhecida como **integridade de produto**. Segundo Bersoff[6], a definição formal de integridade de produto seria um conjunto intrínseco de atributos que caracterizam um produto, de maneira que:

- preencham as necessidades funcionais do usuário,
- possam ser fácil e completamente rastreados através do ciclo de vida,
- satisfaçam os critérios de desempenho estipulados e
- satisfaçam as previsões de custo e prazo.

Logo, o objetivo está claro agora, pois então, cabe ao processo ter como resultado produtos de software que possuam características de integridade de produto. Mas qual caminho nos leva a este objetivo? O processo de desenvolvimento é (ou deveria ser) sustentado por três conjuntos básicos de atividades:

1. atividades de gerenciamento,

2. atividades de desenvolvimento e

3. atividades de gerência da qualidade.

As atividades de gerenciamento devem propiciar toda a infra-estrutura necessária para o bom funcionamento das outras duas atividades.

As atividades de desenvolvimento, numa visão bem geral, são aquelas que fornecem os meios para a construção do software em si, ou seja, são as que auxiliam a transformação de uma idéia em produto final.

As atividades de gerência da qualidade são as utilizadas pelo gerenciamento para obter visibilidade sobre o processo de desenvolvimento, e incluem:

- Gestão de Configuração,

- Garantia de Qualidade, onde estão inseridas as atividades de Verificação, Validação e Teste (VV&T).

Elas são utilizadas para, em determinados pontos do processo, fornecer meios para se certificar de que o que se está construindo é o que se espera, e da maneira que se espera.

Agora, então, podemos responder nossa questão inicial. A gestão de configuração é uma atividade que auxilia a construir produtos de software com garantia de integridade.

Do ponto de vista da qualidade, a gestão de configuração é uma necessidade para adequação às novas normas internacionais sobre qualidade. Segundo o modelo de maturidade de processos CMM/SEI[54], para que uma empresa consiga chegar ao nível 2 do modelo, um dos primeiros passos é ter implementado e institucionalizado a gestão de configuração de software. Apenas a título de exemplo, o Departamento de Defesa Norte Americano só está aceitando, nas suas con-

corrências, empresas classificadas no nível 2 e acima, alegando que empresas com níveis inferiores podem colocar os sistemas de defesa em risco[8].

3.3.3 O Que Controlar?

Os elementos básicos de controle de uma configuração são os ítens de configuração (IC). Um item de configuração é todo e qualquer ente passível de manipulação, ou seja, que possa ser univocamente identificado e gerenciado. São normalmente considerados ICs, documentos e código (seja fonte, executável etc.) e eles são obtidos através de um processo de decomposição que terá como resultado um conjunto de objetos hierarquizados e logicamente inter-relacionados.

Recomenda-se, para se obter uma boa visibilidade do projeto, que todos os produtos gerados em um projeto sejam controlados sob forma de ICs. Neste caso, utilizando-se a MUSA, é recomendável que todos os produtos nela especificados como resultados das diversas fases sejam declarados ítens de configuração.

Vale lembrar que existe um compromisso entre a granularidade, visibilidade e gerenciamento da configuração. Segundo a norma NBR ISO 10007[1], quanto maior a granularidade, mais difícil a visibilidade e maior o esforço de controle (acarretando um maior custo). Um número muito pequeno de ICs (decomposição insuficiente) leva a dificuldades de manutenção e limita as possibilidades de gerenciamento. A escolha do melhor equilíbrio entre essas premissas é de caráter puramente gerencial e deve ser estudada caso a caso. Um critério que poderia ser usado nesta seleção, seria escolher os itens que possuam características físicas e de desempenho que possam ser gerenciadas de maneira isolada.

3.3.4 Como Controlar?

O processo de GCS é composto de várias atividades, alguns autores como Pressman falam em cinco atividades, mas a maioria deles, como[6][7][12][8] e também a norma NBR ISO 10007[1], aceitam a divisão clássica em quatro atividades, que são as seguintes:

1. Identificação de Configuração,
2. Controle de Configuração,
3. Contabilização da Situação de Configuração (também conhecida como *status* da configuração) e
4. Auditoria de Configuração.

Para efeito de esclarecimento, no relatório técnico específico sobre gestão de configuração de software ISO/IEC 15846 [40], a atividade de auditoria de configuração é chamada de avaliação da configuração.

3.3.4.1 Identificação de Configuração

Esta atividade consiste em nomear de maneira única os componentes de uma configuração. Estes componentes básicos são os itens de configuração (IC). Um IC pode ser tanto um trecho de código quanto um documento, e deve ficar a cargo do projeto de controle de configuração definir quais serão os ICs do projeto que serão controlados.

A forma de nomeação dos ICs é completamente livre, mas deve ser padronizada dentro do projeto. Pode-se aproveitar a estrutura do produto (quando ela for hierárquica) ou uma outra forma qualquer. Para a indicação numérica, pode-se adotar, por exemplo, o uso de dois numerais, sendo o primeiro a indicação da configuração e o segundo a indicação da versão do IC. No entanto, hoje em dia esta informação é fornecida automaticamente pelas ferramentas de controle.

O padrão da ABNT/ISO para Gestão de Configuração[1] prevê as seguintes funções dentro dessa atividade:

1. **Seleção dos itens de configuração:** Os ICs devem ser escolhidos pelo processo de decomposição.

2. **Determinação da estrutura da configuração:** Esta estrutura deve descrever a posição de cada IC que compõe o projeto. O uso de uma estrutura hierárquica é recomendável.

3. **Documentação dos ítens de configuração:** Todas as características físicas e funcionais de um item (interfaces, alterações, desvios e concessões (*waivers*)) devem ser descritas em documentos bem identificados.

4. **Sistema de numeração:** Um sistema de identificação deve ser escolhido para garantir unicidade na identificação dos ICs.

5. **Estabelecimento de configurações básicas:** Configurações básicas devem ser estabelecidas através de um acordo formal em determinados pontos do ciclo de vida do projeto e utilizadas como ponto de partida para o controle formal da configuração. Estes pontos podem ser as mudanças de fase no ciclo de vida, bem como marcos estabelecidos no cronograma do pro-

jeto.

3.3.4.2 Controle de Configuração

Segundo Bersoff[7] o conceito de configuração-base (*baseline)* é um dos fundamentos da Gestão de Configuração. A geração de uma configuração-base é o momento no qual é acordado que um ou mais IC estão em conformidade com os requisitos do projeto e devem ser protegidos contra alterações não autorizadas. Após o estabelecimento de uma configuração-base, toda e qualquer alteração sobre os itens dessa configuração deve ser **formalmente controlada**.

O impacto da alteração, os requisitos do cliente e a configuração-base afetada influenciam o grau de formalidade do processo de alteração e podem servir de base para qualquer sistema de classificação utilizado para a classificação/categorização de alterações.

As seguintes atividades devem ser documentadas em detalhes no processo de controle:

- Documentação e justificação da alteração.
- Avaliação das conseqüências da alteração.
- Aprovação/recusa dos pedidos de alteração.
- Implementação e verificação das alterações.
- Desvios e concessões (*waivers*).

A fim de proteger a integridade da configuração e fornecer a base para o controle de alteração, é essencial que os ICs sejam mantidos em um ambiente que:

- Satisfaça as condições ambientais necessárias (por exemplo, para hardware, software, dados, documentos, diagramas.).
- Proteja-os de alterações não autorizadas e destruição acidental.
- Forneça mecanismos para recuperação.
- Permita recuperação controlada de todos os dados armazenados.
- Proporcione consistência entre a configuração construída e a especificada.

3.3.4.3 Contabilização da Situação de Configuração

A obtenção de informações sobre o *status* da configuração deve começar assim que os ICs sejam gerados. Estas informações devem permitir o rastreamento de todas as alterações efetuadas sobre os ICs. Desta maneira, podemos saber quem efetuou uma alteração, quando ela foi executada e o que foi feito, por exemplo.

3.3.4.4 Auditoria de Configuração

Segundo Bryan[11] e o próprio IEEE[36], auditoria é o processo de examinar um produto, sendo que esse processo é executado por um grupo independente do grupo que o produziu. Bryan também sustenta que quanto mais crítico é o projeto em termos de tamanho, custo ou cronograma, maior é a necessidade de se ter um processo eficiente de auditoria.

Podemos dizer que auditoria é o preço da qualidade no projeto, no sentido de que através dela é assegurado ao usuário que o produto (no caso o software) está de acordo com os requisitos de desempenho e operacionais, ao cliente que será entregue no prazo especificado e dentro dos limites orçamentários e, por fim, ao fornecedor que seu produto evoluiu de uma maneira rastreável (e portanto manutenível) e de acordo com as expectativas do usuário final.

A auditoria de configuração deve ser executada antes que uma configuração-base seja definida, para certificar que o produto está de acordo com os requisitos (de especificação e contratuais) e também para garantir que está precisamente descrito em seus documentos técnicos. Toda alteração também precisa ser auditada para garantir a integridade do produto que está sendo produzido. Ou seja, ela é utilizada para tornar visível ao gerenciamento o *status* do software ao longo do ciclo de vida do projeto e também para revelar se os requisitos iniciais estão sendo satisfeitos e se a passagem de uma configuração a outra não descaracteriza o produto.

Duas atividades são executadas neste processo:

- **Verificação:** Atividade que assegura a coerência entre as configurações, ou seja, toda configuração candidata a se tornar uma nova configuração-base deve ser verificada com relação à configuração-base anterior.
- **Validação:** Atividade que determina a coerência entre as várias configurações-base e a especificação de requisitos, ou seja, ela garante que cada configuração-base, em cada estágio do ciclo de vida, está de acordo com os requisitos especificados.

3.3.5 Quem Deve Gerenciar?

Os recursos humanos básicos necessários para o gerenciamento de configuração são[8]:

- um responsável para coordenar as atividades de controle,
- um responsável pela biblioteca de ítens sob controle de configuração e
- um Conselho de Configuração (CoC) que tem o poder de: autorizar ,ou rejeitar, alterações, atribuir recursos para a execução das alterações aprovadas e aprovar liberação de componentes de software.

De acordo com o Instituto de Engenharia de Software da Universidade de Carnegie Mellon[64], o trabalho do CoC é sustentado por quatro princípios fundamentais:

1. **Princípio da autoridade:** O CoC tem de possuir autoridade para avaliar, aceitar ou recusar um pedido de alteração sobre um produto sob sua responsabilidade.
2. **Princípio da responsabilidade centralizada:** As decisões do CoC devem ser tomadas por uma única pessoa, após reunião com os membros do CoC. O processo não é democrático.
3. **Princípio da especificidade:** O escopo de responsabilidade do CoC deve ser muito bem definido.
4. **Princípio da pronta resposta:** O CoC deve responder o mais rápido possível a toda solicitação recebida. Um aviso de que a solicitação foi recebida é também muito importante.

Suas características principais são:

1. **Hierarquia do CoC:** Os conselhos são estabelecidos para o sistema como um todo, para os subsistemas e para os componentes do produto. Este conceito de hierarquia implementa o **Princípio da especificidade**.
2. **Escopo de autoridade:** Para implementar o **Princípio da autoridade**, o escopo de itens pelos quais cada CoC é responsável deve ser claramente definido. A documentação sobre o escopo de autoridade de cada CoC deve fazer parte do **Plano de Gestão de Configuração**.
3. **Composição:** Para implementar o **Princípio da responsabilidade centralizada** é necessário que cada CoC tenha uma pessoa encarregada de tomar as decisões em nome do conselho. É

interessante, também, que os níveis de controle estejam de acordo com o nível do projeto. Usualmente, o conselho deve ser formado pelos gerentes de desenvolvimento, de especificação, de análise, de projeto, de manutenção, de produção e por representantes do cliente.

O CoC deve ser um grupo bastante ativo, com um calendário de reuniões regulares, mas, também, deve ter agilidade suficiente para tratar de alterações de emergência. O procedimento de emergência está muito bem explicado em Marian[8].

Suas atividades incluem o estabelecimento do calendário de entrega de produtos de software, a autorização ou recusa de pedidos de alteração e o estabelecimento de seu procedimento de trabalho. Para as tomadas de decisão, o CoC deve levar em consideração os seguintes fatores, entre outros:

- extensão da alteração,
- complexidade da alteração em relação ao sistema,
- data da finalização,
- impacto no trabalho atual e nos futuros,
- custo da alteração,
- criticalidade da área envolvida,
- *status* das alterações aprovadas,
- requisitos de teste,
- recursos envolvidos (habilidades, software, hardware etc.),
- impacto sobre uso de CPU e de memória,
- políticas relevantes,
- maturidade da alteração e
- verificação de possíveis alternativas.

Para saber quais desses itens devem ser considerados, deve-se observar se o item analisado é uma inconsistência, um pedido de alteração ou uma melhoria.

3.3.6 Quando Controlar?

O controle efetivo de um IC deve começar a partir do momento em que as características deste item sejam aceitas como corretas naquele momento. Isto significa que o item passa, a partir de então, a sofrer um controle formal de alterações, em que toda e qualquer alteração pretendida deve ser aprovada, executada, revisada e documentada.

O conceito aqui utilizado é o de configurações-base (*baselines*). Essas configurações nada mais são do que uma representação momentânea do que está valendo como base (aprovada) daquele estágio do desenvolvimento. É comum utilizar-se o fim das fases do ciclo de vida de desenvolvimento como marcos para o estabelecimento destas configurações-base, mas nada impede que, para cada projeto, outros marcos sejam escolhidos.

No caso da utilização da MUSA, recomenda-se que ao fim de cada fase seja realizada uma revisão (para validação e verificação) dos produtos gerados naquela fase e, no caso de haver acordo, com relação à forma e conteúdo desses produtos, uma configuração-base seja criada, colocando esses produtos sob controle formal de alterações. Deste modo, de acordo com a MUSA, podemos identificar, no mínimo, as seguintes configurações-base:

1. **Configuração-Base de Especificação:**

   Estabelecida após a Fase de Especificação de Requisitos.

2. **Configuração-Base Funcional:**

   Estabelecida após a Fase de Projeto Lógico.

3. **Configuração-Base Descritiva:**

   Estabelecida após a Fase de Projeto Físico.

4. **Configuração-Base de Produto:**

   Estabelecida após a Fase de Implementação.

3.4 Considerações finais

Neste capítulo, foram discutidos o teste e a gestão de configuração de software. Foi mostrada a necessidade de se testar, foram apresentados os vários níveis de teste (unidade, integração,

sistema e aceitação) com seus objetos de teste e uma breve discussão sobre o ambiente desses testes. As três técnicas de teste mais conhecidas e seus critérios associados foram apresentados:

- a técnica funcional: critérios particionamento de equivalência, análise do valor limite, grafo de causa e efeito;
- a técnica estrutural: critérios baseados em fluxo de controle (todos-os-nós, todos-os-arcos, todos-os-caminhos, cobertura de decisão e cobertura de condição); baseados em fluxo de dados (todas-as-definições, todos-os-usos e potenciais-usos) e teste de laço, proposto por Boris Beizer e
- a técnica baseada em erros: critério análise de mutantes.

E para finalizar a discussão sobre teste foi apresentado um conjunto de documentos para apoio ao processo de teste.

Sobre o processo de gestão de configuração de software foram mostradas a necessidade da gestão de configuração, suas funções básicas (identificação da configuração, controle da configuração, contabilização da situação da configuração e auditoria da configuração), seus benefícios, as normas internacionais que apóiam o processo e a estrutura de controle. Discutiu-se sobre configurações-base (*baselines*) e foram citadas várias ferramentas que automatizam o processo. No próximo capítulo, será discutida a questão de apoiar a atividade de teste com o auxílio do processo de gestão de configuração. Serão tratados a importância da informação de teste, o volume de dados gerados na atividade de teste, a dificuldade de gestão desses dados e os benefícios de se utilizar a gestão de configuração para esse controle. Será apresentada uma maneira de se caracterizar um item de configuração para teste e, então, será apresentada uma proposta de solução com a especificação de um ambiente de gestão de configuração de software para teste.

Capítulo 4

Proposta de uma abordagem para a gestão de configuração do teste de unidade de software

4.1 Considerações iniciais

A atividade de teste de software abrange atividades similares às atividades de desenvolvimento de software, pois o testador deve planejar a atividade, projetar e construir os elementos necessários ao teste, executar os testes e colher e analisar os resultados.

Pelo fato do foco não ser o modelo de desenvolvimento usado, o teste não será visto como sendo uma fase ou algo parecido, mas apenas como uma atividade na qual o programador/testador, ao terminar o desenvolvimento de uma unidade, deve exercitá-la para validar o resultado do desenvolvimento. A atividade de teste é, então, caracterizada pela existência de várias sessões de teste, sendo que uma sessão de teste engloba o planejamento dos testes, o projeto dos casos de teste, a execução dos testes e a análise dos resultados.

No planejamento, o programador vai escolher os critérios de geração e adequação dos casos de teste e os de encerramento da atividade de teste, ou seja, vai determinar de que maneira os casos de teste serão gerados (aleatório, por exemplo), como eles serão analisados (fluxo de dados, por exemplo) e como será decidido o encerramento da atividade em si (o programa executa como esperado? o número de erros encontrados está convergindo?). Aqui, o programador também decide sobre a disponibilidade de ambiente e ferramentas (se precisam ser adquiridas, desenvolvidas, se é necessário treinamento especial, se é preciso pessoal especializado para instalação etc.). Como produto do planejamento, é gerado um documento chamado **Plano de Teste**, simples, mas com informações relevantes para a gestão da atividade. Neste documento, deveriam estar contidas informações como:

- Identificação do programador/testador,
- Objetivo do teste,
- Data,
- Identificação do programa testado,
- Funções a serem testadas,

- Critérios de geração e adequação dos casos de teste e de encerramento da atividade de teste,
- Ambiente de teste (versões de ferramentas, versão de SO etc.),
- Conjunto de casos de teste,
- Diagramas de projeto e
- Procedimento de teste.

No projeto dos casos de teste, o programador/testador aplica o critério de geração dos casos de teste (aleatório, funcional etc.), determina os dados de teste e os resultados esperados; e desenvolve os programas de apoio ao teste, ou seja, programas que ativem o programa em teste (também chamados de *drivers*) e programas que simplesmente respondam a estímulos do programa em teste (também chamados de *stubs*).

Durante a execução, o programador exercita o programa em teste com os casos de teste projetados. Aqui, além de informações de execução como resultado real, tempo de execução e falhas encontradas, o programador/testador obtém informações dependentes do(s) critério(s) adotado(s). Caso seja adotado um critério baseado em fluxo de dados, serão obtidas as associações e os caminhos cobertos. No caso da análise de mutantes, serão obtidos os mutantes mortos, os vivos e os equivalentes. Se o critério escolhido for fluxo de controle, serão obtidos os nós (ou arcos) cobertos e os não cobertos, por exemplo. No caso da atividade ser automatizada, é possível, também, comparar os resultados obtidos com os esperados.

Na análise, o programador analisa a cobertura obtida pelo conjunto de teste, segundo o critério de adequação adotado, com o objetivo de completar o conjunto de teste de acordo com os requisitos de teste ainda não cobertos. Compara os resultados obtidos com os esperados para fins de validação da atividade e registra os problemas encontrados. No final, é gerado um documento chamado **Relatório de Teste** com as seguintes informações:

- Identificação do testador,
- Identificação do Plano de Teste associado,
- Resultados obtidos e problemas encontrados,
- Análise de adequação,

- Validação e
- Data de encerramento do teste.

Informações advindas dos resultados do teste, como por exemplo uma alta cobertura para um dado conjunto de casos de teste gerados por um ou mais critérios conhecidos pela capacidade de revelar erros, auxilia no aumento da confiança sobre o software.

No entanto, o volume de informações gerado durante o teste torna a atividade de teste bastante complexa e de difícil controle, caso isso não seja feito de forma sistemática e controlada. Uma das maneiras de introduzir esse controle sistemático à atividade de teste é utilizar os conceitos de gestão de configuração para administrar a atividade de teste. A proposta de usar a disciplina de gestão de configuração está focada no problema de gerenciar grandes volumes de dados, pois ela vai auxiliar a organizar todo esse trabalho, de forma a garantir a integridade de todos os produtos gerados e, principalmente, a rastreabilidade a todos eles.

Num espectro macro, pode-se imaginar que os grandes conjuntos a serem gerenciados são: os documentos de apoio à atividade de teste, o software a ser testado, os *drivers* e *stubs* criados, os requisitos de teste e os casos de teste, os resultados obtidos e a análise feita com base nos resultados obtidos. No entanto, existe um ciclo de teste, retorno ao desenvolvimento para correção ou introdução de melhoramentos e novo teste, até que o programa seja liberado. Isto envolve a existência de várias versões de desenvolvimento do programa em teste e do respectivo conjunto de casos de teste (e, claro, dos resultados obtidos). Todo este histórico é de suma importância ao pensarmos no teste de regressão[70][27], ou seja, na necessidade de garantir que as alterações sofridas pelo software não o alteraram de tal forma que ele não mais execute corretamente um teste mais antigo, pelo qual já havia passado com sucesso. É importante, também, ter-se em mente que, após a liberação, cada pacote entregue pode vir a sofrer manutenções diferentes e constituir versões diferentes do mesmo programa. Imaginemos o caso de grandes sistemas, desenvolvidos para as prestadoras de serviço de Telecom, que já são especificados para as peculiaridades de cada prestadora, com sistemas da ordem de milhões de linhas de código. Mesmo que parte do código seja comum a todos eles, a parte que pertence a apenas um sistema já demanda um número muito grande de módulos com várias versões que deverão ser geridos, isso sem falar nos testes específicos, que são associados a esses módulos (e suas versões) e que também devem ser administrados.

A gestão de configuração vai auxiliar quanto à unicidade de nome dos elementos, quanto ao controle das alterações sobre esses elementos e na criação de ligações entre esses elementos de uma maneira lógica, para garantir seu gerenciamento. Por exemplo, a um módulo estariam ligados seu conjunto de casos de teste e seu conjunto de saída, sendo que, para cada versão necessária (devido

às correções sofridas pelo software), seria armazenada apenas a diferença entre os conjuntos, diminuindo assim o espaço necessário a esse armazenamento.

O objetivo, agora, é discutir os elementos envolvidos em um teste de unidade e baseado nas funções de identificação e de controle da atividade de Gestão de Configuração, propor qual deveria ser a composição de um item de configuração de teste (ICT) e como seria a integração dos ambientes de teste e gestão de configuração. Analisamos o que é importante para um item de configuração de teste, quais são os conjuntos candidatos, o que deve ser armazenado e o que pode ser regerado. São também apresentadas quais funções que dariam suporte à automatização das atividades de gestão de configuração para os itens de configuração de teste.

4.2 Um Modelo de Item de Configuração de Teste

Basicamente, um item de configuração de teste deve conter os elementos necessários à reexecução de uma dada sessão de teste, ou à observação de seus resultados a posteriori, sem com isso causar algum tipo de distúrbio no ambiente de teste atual, seja em termos de retrabalho ou de tempo despendido para recriar o ambiente e reexecutar os testes. Com isto, pode-se supor que o critério básico para a escolha de candidatos a compor o ICT é a escolha de informações importantes e que são custosas de serem regeradas.

Pode-se, então, supor que o testador deveria armazenar as informações relativas ao planejamento do teste e projeto dos casos de teste, já que toda a motivação do teste, bem como as informações complementares, saem do planejamento e o projeto dos casos de teste é uma atividade que demanda muito tempo (caso não seja usada uma ferramenta automática para isto). Na execução dos testes são realizadas as análises estáticas e dinâmicas, o programador deveria armazenar as informações de execução geradas pelas ferramentas de teste (análise dinâmica), já que demandam a execução do programa com todos os casos de teste e isto, às vezes, chega a demorar dezenas de horas. Como as informações vindas da análise estática do programa são facilmente regeradas, então, não necessitam ser guardadas. Os resultados obtidos da fase de análise dos resultados da atividade de teste também deveriam ser armazenados.

Com isto, pode-se ter uma idéia mais clara de que o item de configuração de teste deveria ser composto de:

- plano de teste (quando houver),
- conjunto de casos de teste,
- *drivers* e *stubs* utilizados,

- relatório de teste (quando houver),
- informações de execução relativas a cada critério de adequação utilizado (estas informações são dadas pelas ferramentas utilizadas) e
- informações de análise dos resultados dos testes.

Observe que um dos componentes do ICT é o conjunto de informações relativas ao critério de adequação utilizado. É justamente nesse ponto que está a abertura para a caracterização do ICT independente do critério. Ou seja, pode-se caracterizar ICTs para teste funcional, estrutural ou baseado em erros. Até mesmo para novos critérios, para tanto basta conhecer que informações de execução são geradas, quais são mais importantes para serem armazenadas e qual é a estrutura utilizada pela ferramenta de teste para armazenar essas informações.

4.3 Ambiente de gestão de configuração de software para teste

O ciclo de gestão de configuração para teste é mostrado na Figura 1. O testador deve configurar o ambiente, fazer a integração da ferramenta de teste a ser utilizada e então criar uma sessão de teste. A essa sessão de teste (ST) são associadas uma ou várias sessões de análise de cobertura[1] (SAC) onde os testes são executados e os resultados coletados e analisados. Obtendo o resultado esperado elas poderão ser salvas, gerando as configurações-base das várias SACs. Outras sessões de teste poderão ser criadas, ou então o teste pode ser encerrado. A qualquer momento outras ferramentas de teste podem ser acrescentadas, ou excluídas.

O modelo de operação do ambiente de gestão de configuração para teste está baseado no modelo de dados visto na Figura 2. Caso o programador/testador tenha um plano de teste, este plano implementa uma única estratégia de teste, ou seja, a maneira como os testes serão conduzidos no que diz respeito ao uso dos critérios e das ferramentas.

Uma função de um programa pode ser testada por mais de uma sessão de teste. Uma estratégia (independentemente do plano) está relacionada a uma ou várias ST. Uma ST contém uma ou várias sessões de análise de cobertura (SAC), sendo que uma sessão de análise de cobertura é feita com o apoio de uma ferramenta e um critério de cobertura apoiado por essa ferramenta.

1. O conceito de SAC foi criado por motivos de implementação do ambiente, já que dessa forma era possível relacionar uma única ferramenta de teste a um critério apoiado por essa ferramenta. Permitindo, dessa forma, que numa mesma sessão de teste (ST) um critério fosse utilizado por mais de uma ferramenta de teste.

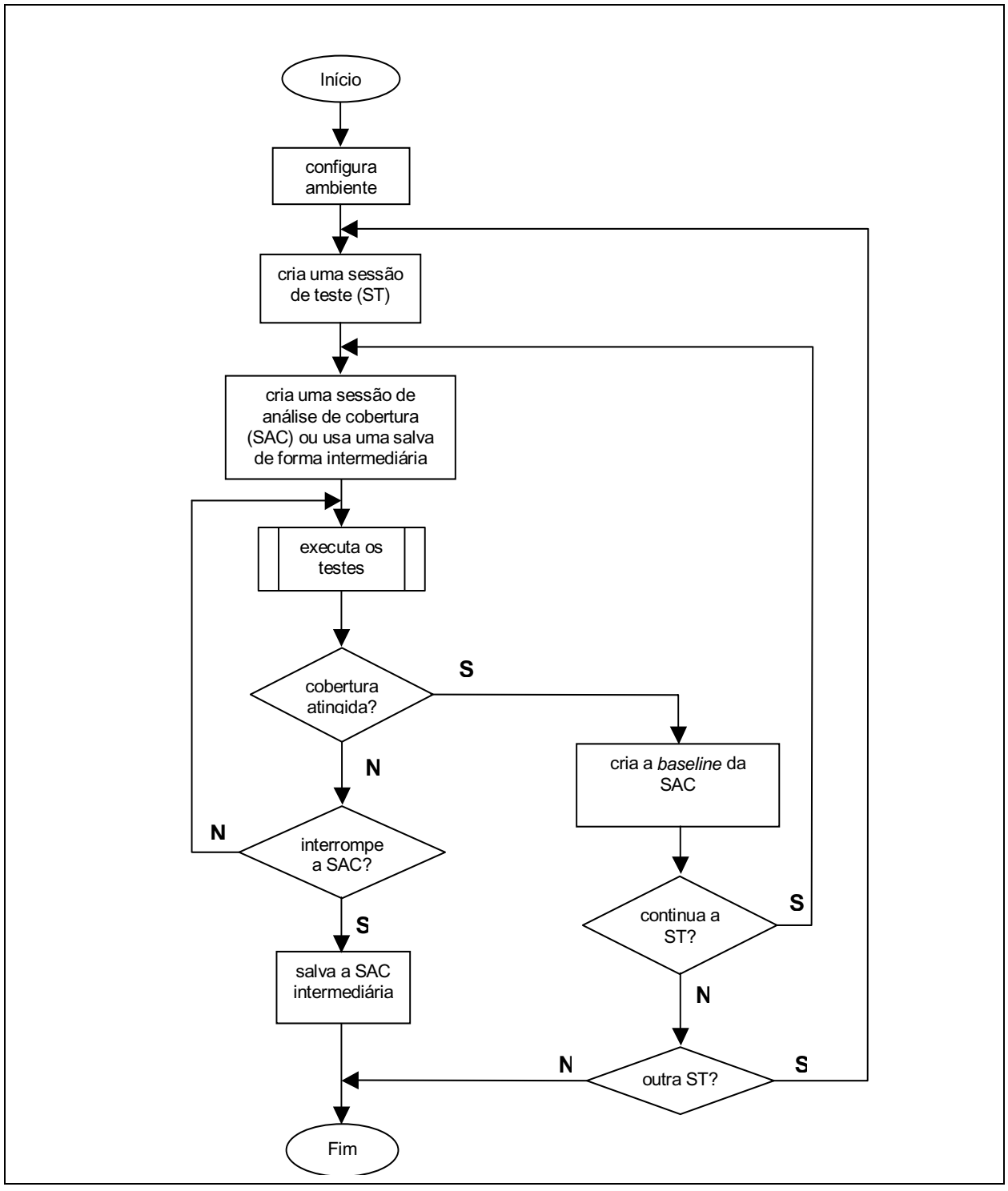

Figura 1 - Ciclo de gestão de configuração para teste

Com essas informações, um ambiente de gestão de configuração de software para teste deve:

- Permitir integração com qualquer ferramenta de teste (isso implica em reconhecer a estrutura de dados implementada pela ferramenta de teste).

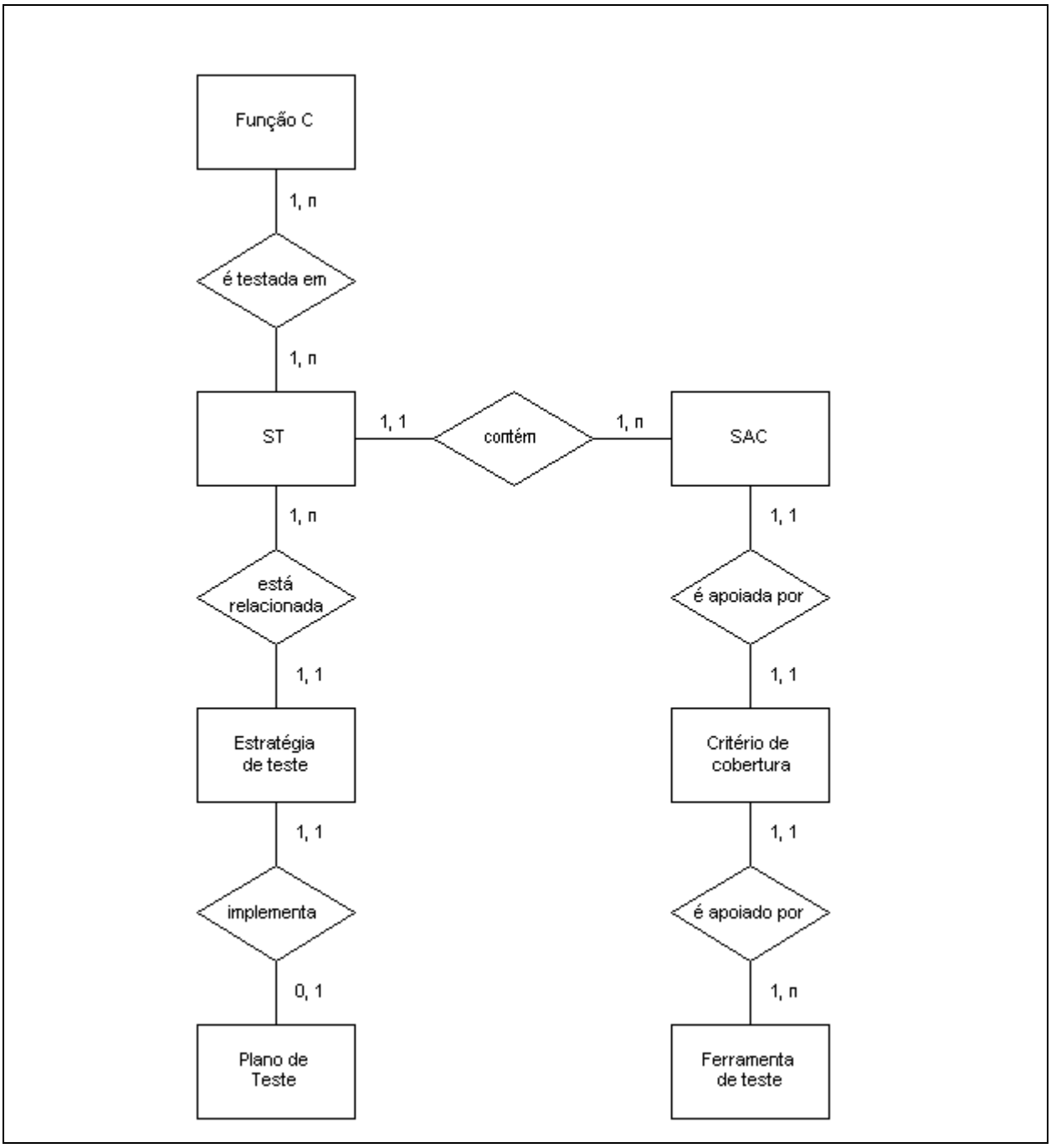

Figura 2 - Modelo de dados teste

- Permitir a caracterização do item de configuração de teste,
- Permitir inclusão/exclusão de uma ferramenta de teste,
- Permitir inclusão/exclusão de critérios associados às ferramentas de teste,
- Permitir inclusão/exclusão de sessões de teste,

- Permitir a inclusão/exclusão de sessões de análise de cobertura,
- Permitir salvamento/recuperação intermediário das informações de teste e
- Permitir salvamento/recuperação integral das informações de teste (geração/recuperação de configurações-base).

Vejamos agora uma descrição do funcionamento do ambiente de gestão de configuração para teste.

4.3.1 Configuração do ambiente de software

O primeiro passo do testador é a preparação do ambiente de software, que nesse caso consiste da criação dos diretórios de trabalho e da declaração das variáveis de ambiente UNIX. Precisam ser criados os diretórios: TESTDIR (o diretório de teste), CONFDIR (o diretório onde está a estrutura de integração com as ferramentas de teste), LIBDIR (o diretório onde estarão as sessões de teste) e um diretório onde residem os comandos implementados.

4.3.2 Integração com as ferramentas de teste

Para integrar as ferramentas de teste, o testador deve configurar o ambiente de GCST, onde serão inseridas informações sobre:

- As ferramentas usadas;
- Os critérios apoiados;
- Os elementos do ICT de cada critério, com:
- Nome do arquivo associado,
- Endereço do arquivo e
- Uma breve descrição do IC.

Essa configuração de ambiente é garantida por meio de uma estrutura de diretórios (veja Figura 3) na qual abaixo do diretório pai chamado **conf** é criado um diretório para cada ferramenta de teste utilizada; abaixo do diretório de cada ferramenta de teste é criado um diretório para cada critério apoiado por essa ferramenta. Dentro do diretório de cada critério, é criado um arquivo (**ArqIC**) que contém as informações sobre os elementos do ICT desse critério; essas informações estão estruturadas em conjuntos de 3 linhas nas quais, na seqüência, estão: o nome do elemento do ICT, o endereço em que o arquivo deste elemento do ICT é encontrado e uma breve descrição.

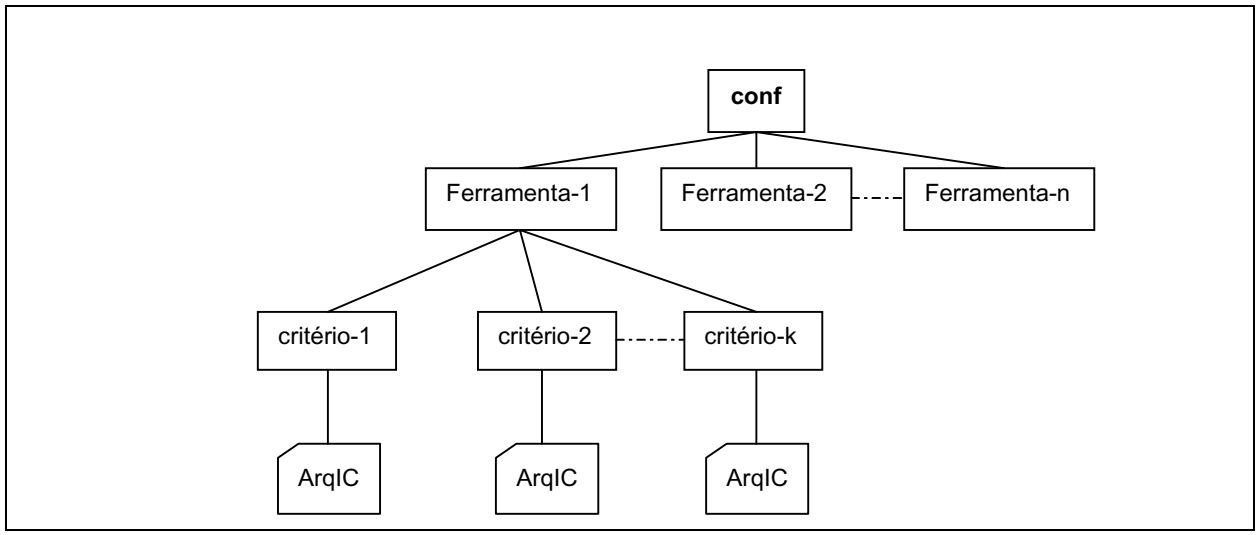

Figura 3 - Estrutura de diretórios de configuração do ambiente de GCST

Arquivo ArqIC

<nome do arquivo-A>

<endereço do arquivo-A>

<descrição do arquivo-A>

...

Essa configuração é feita com o uso de um comando (**qconfigura**) interativo que pergunta sobre as informações pertinentes, coleta as respostas e monta a estrutura de diretórios descrita acima. Dessa forma, o ambiente de GCST será capaz de saber onde e quais informações deverá capturar ao final das várias sessões de teste e de análise de cobertura. Essa forma de configuração faz com que o ambiente de GCST seja aberto à integração com novas ferramentas de teste, permitindo a inclusão e exclusão delas. Para quaisquer alterações na configuração do ambiente de gestão de configuração para teste, o mesmo comando deve ser usado.

4.3.3 Criação de uma sessão de teste

Após a fase de configuração, o testador deve cadastrar as ST. Isso é feito com o uso de um comando (**qcriast**) interativo que primeiramente obtém o nome da ST e, caso ela já tenha sido cadastrada, avisa o testador. Sendo uma nova ST, o comando irá criar, abaixo do diretório das sessões de teste, um subdiretório com o nome da ST. Nesse subdiretório, será criado o sub-diretório de trabalho da ST (**WORK**) o qual conterá o repositório controlado (repositório RCS). No subdiretório da ST, por meio das respostas obtidas do testador, o comando irá criar 2 arquivos a saber: **ArqST** e **ConfST** (veja na Figura 4). O arquivo **ArqST** tem na primeira linha o nome do testador, nas linhas subseqüentes, ele possui um conjunto de informações sendo uma linha com o

nome da ferramenta e uma, ou mais linhas com o nome dos critérios apoiados, seguido do nome da ferramenta, por exemplo:

Arquivo ArqST

<nome do testador>

<ferramenta X>

<critério a> <ferramenta X>

<critério b> <ferramenta X>

<ferramenta Y>

<critério c> <ferramenta Y>

O arquivo **ConfST** só existe se houver documentos de teste relacionados à ST (plano de teste, casos de teste e registro de teste). Neste caso, o arquivo possuirá um conjunto de informações que estarão estruturadas em um conjunto de 3 linhas nas quais, na seqüência, estão: o nome do elemento do ICT, o endereço onde esse arquivo é encontrado e uma breve descrição.

Arquivo ConfST

<nome do arquivo-A>

<endereço do arquivo-A>

<descrição do arquivo-A>

...

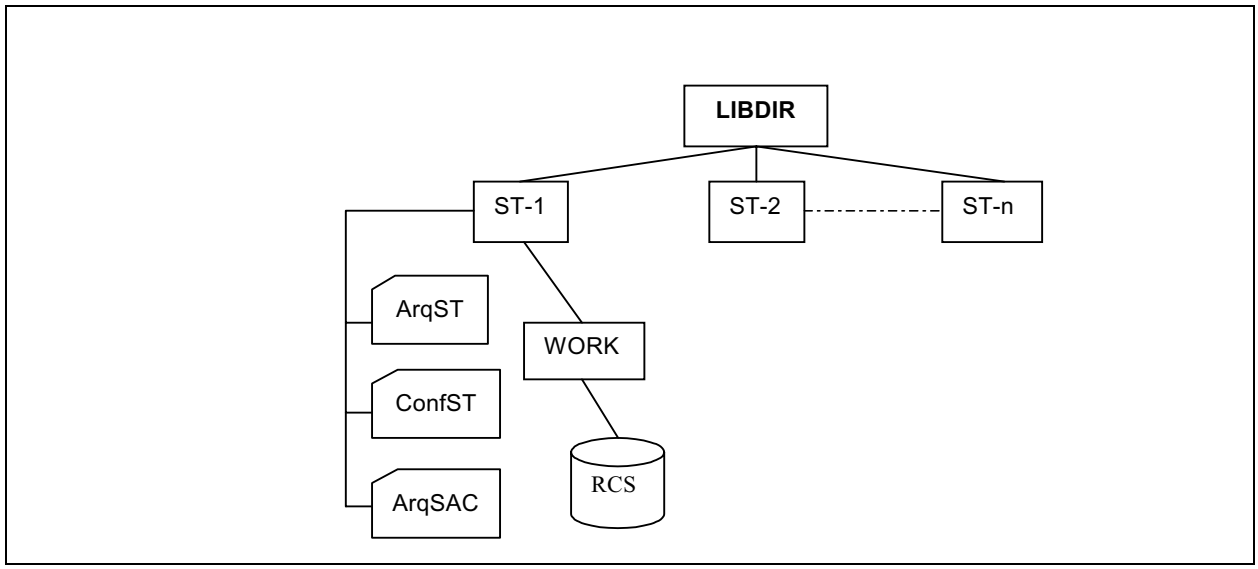

Figura 4 - Diretório das sessões de teste

4.3.4 Criação de uma sessão de análise de cobertura

Após o cadastramento da ST o testador já pode também cadastrar as SAC que fazem parte dessa ST. O cadastro da SAC é feito com o auxílio de um comando (**qcriasac**) que recebe como parâmetros: o nome da SAC, o nome da ST associada, o critério apoiado e a ferramenta usada. Ele gera o arquivo **ArqSAC** que contém informações sobre todas as SAC cadastradas em determinada ST (veja Figura 4). Cada linha desse arquivo é formada, na seqüência, pelo nome da SAC, da ferramenta usada e do critério apoiado, por exemplo:

Arquivo ArqSAC

<sac K> <ferramenta X> <critério a>

<sac W> <ferramenta Y> <critério c>

...

4.3.5 Salvamento intermediário

Após a fase de configuração do ambiente, o testador executa os testes propriamente ditos, exercitando a função em teste com os dados de teste elaborados, até o momento em que a sessão de análise de cobertura (SAC) é interrompida e o testador analisa a cobertura obtida (de acordo com o critério escolhido). Caso a cobertura esperada não tenha sido atingida, ele pode decidir continuar a execução dos testes até atingi-la. No entanto, se por qualquer motivo o testador decide interromper seu trabalho e quer proteger os dados obtidos até o momento, ele pode criar uma versão intermediária da SAC com o uso de um comando (**qsalvasac**) que recebe como parâmetros o

nome da SAC, o nome da ST e o motivo da criação da versão. Esses dados são validados e se a SAC ou ST não forem válidas o testador é avisado por uma mensagem de erro. Sendo válidas, uma versão da SAC é criada no repositório RCS do diretório de trabalho (**WORK**) da ST em questão e o testador poderá continuar seu trabalho posteriormente, não necessitando recuperar explicitamente esta versão, pois a ferramenta faz isso automaticamente.

4.3.6 Criação de uma configuração-base

Quando a cobertura esperada é atingida, uma configuração-base da SAC pode ser criada e isso é feito por meio do uso de um comando (**qcongelasac**) que recebe como parâmetros o nome da SAC e o nome da ST. Caso alguma delas não esteja cadastrada, o testador é avisado por uma mensagem de erro. Em caso positivo, pelo nome da ST e da SAC, o comando recupera a ferramenta e o critério usados; com esses dados o comando pode obter as informações sobre os elementos do ICT dessa SAC (fazendo acesso aos dados relacionados à ST e à SAC contidas na estrutura de configuração, abaixo do diretório **conf**). Com essas informações (nome, endereço e descrição dos arquivos do ICT), é possível criar uma configuração-base com o nome da SAC, na qual esses dados estarão protegidos contra qualquer alteração não desejada, ou seja, esses dados estarão disponíveis apenas para leitura e, portanto, apenas para análise. Essa configuração-base é localizada no repositório RCS, abaixo do diretório **WORK** do diretório da ST em questão. Podem ser criadas tantas configurações-base de SAC quantas forem as SAC cadastradas na ST.

4.3.7 Recuperação de uma configuração-base

Para efeito de análise, o ambiente de GCST apresenta o comando **qrecuperasac** para recuperação da informação associada a uma SAC. O comando **qrecuperasac** recebe como parâmetros o nome da SAC e o nome da ST que, depois de validados, recoloca os dados do ICT da SAC nos diretórios correspondentes da ferramenta de teste. Dessa forma, o testador tem a possibilidade de, a qualquer momento, reanalisar uma sessão já realizada, mas salva na ferramenta da GCST.

A qualquer momento, o testador pode cadastrar novas ferramentas e seus critérios associados, ou retirá-las do cadastro. Novas ST podem também ser criadas, a qualquer momento, com suas SAC associadas.

4.4 Considerações finais

Nesse capítulo, foi discutida a questão do apoio do processo de gestão de configuração à atividade de teste de software. Foram ressaltados dois aspectos: importância da informação de teste e a do uso da gestão de configuração para gerenciá-la, uma vez que esse gerenciamento é dificultado pelo alto volume dessa informação. Explicou-se o conceito de sessão de teste, em que o teste é visto como um desenvolvimento em si, com fases de planejamento, projeto de casos de teste, execução dos casos de teste e análise dos resultados. Em função do volume de dados produzidos

com a automatização da atividade de teste e levando em consideração o custo de obtenção desses dados para análises futuras dos resultados dos testes, foram propostos um modelo de item de configuração de teste e um ambiente para controle desses dados. Foram especificadas as funções básicas do ambiente de gestão de configuração de software para teste, o modelo de dados implementado, o ciclo de controle utilizado e apresentada a especificação funcional do ambiente. No próximo capítulo serão apresentadas a ferramenta de teste Poke-tool, a ferramenta UNIX RCS e o ambiente de gestão de configuração de software para teste, o qual garante a integração entre os dados gerados pela ferramenta de teste e a ferramenta de controle de configuração, bem como, a automatização da coleta dos elementos do ICT. Também será visto um exemplo prático, com a caracterização de um item de configuração de teste real (utilizando o modelo apresentado neste capítulo) e a utilização do ambiente de gestão de configuração de software para teste com o objetivo de controlar a massa de dados gerados durante os testes.

Capítulo 5

Automatização do processo de gestão da informação de teste

5.1 Considerações iniciais

Neste capítulo será apresentada a solução de automatização do ambiente de GCST. Descrevem-se a ferramenta de teste Poke-tool, a ferramenta RCS, o programa **`prime.c`** que é utilizado no exemplo e o ambiente de GCST, desenvolvido para integrar os ambientes de teste e de gestão de configuração. Também é mostrado um exemplo prático de utilização do ambiente proposto no capítulo anterior, configurando o ambiente de software, integrando a ferramenta de teste ao ambiente de GCST, executando os testes, criando as configurações-base e recuperando uma dada configuração-base para a estrutura de dados da ferramenta de teste.

5.2 Ferramentas utilizadas

Nesse trabalho, foram escolhidas duas ferramentas para o desenvolvimento da parte prática. A parte de gestão de configuração está apoiada no uso da ferramenta RCS[63] e a parte de teste utiliza a ferramenta Poke-tool[16][17]. A integração entre esses dois ambientes é feita pelo ambiente de GCST, que permite a caracterização do ICT (para a ferramenta de teste em uso) e controla todas as alterações sofridas ao longo da atividade de teste.

5.2.1 O sistema RCS

O RCS é um sistema de controle de versões (revisões) desenvolvido por Walter Tichy (da Universidade de Purdue) em 1985. Ele é baseado em um conjunto de comandos UNIX, mas também possui versão para ser utilizado em PCs.

A sua função principal é controlar grupos de revisão. Um grupo de revisão é um conjunto de arquivos (chamados revisão) que mantêm uma seqüência cronológica entre si, e que representam a seqüência de evolução do arquivo inicial. A primeira revisão é chamada de raiz.

Os arquivos são armazenados em diretórios do próprio sistema; portanto, a segurança está ligada à segurança no nível dos diretórios. Pode ser criado um subdiretório RCS (onde residirão os arquivos controlados) para facilitar o trabalho, mas isso não é obrigatório.

5.2.1.1 Identificação automática

É possibilitada a identificação automática dos arquivos controlados pelo RCS através do uso de *keywords* colocadas no arquivo fonte em forma de comentário. Por exemplo, o identificador **Id** é expandido para apresentar informações sobre: nome do arquivo, número da revisão, data e hora da revisão, autor da revisão e *status*. O identificador **Log** acumula todas as mensagens de *log*. Caso não se decida pela utilização das *keywords*, o sistema armazena automaticamente algumas informações importantes para o controle.

5.2.1.2 Comandos RCS

Esses são os comandos básicos do RCS, mas vale lembrar que nem todas as implementações UNIX apresentam todo esse conjunto da forma aqui descrita.

1. **ident**: Extrai as informações contidas nos arquivos advindas do uso de *keywords* como Id ou Log.
2. **rcs:** Altera atributos de um arquivo RCS.
3. **rcsclean:** Remove arquivos da área de trabalho que não foram alterados. (opcional)
4. **rcsdiff:** Compara revisões.
5. **rcsfreeze:** Registra (congela) um configuração. (opcional)
6. **rcsmerge:** Faz a união (*merge*) de revisões.
7. **rlog:** Obtém informações do *log* do arquivo.
8. **co:** Retira um item do RCS e o disponibiliza para o programador.
9. **ci:** Coloca um item sob controle do RCS.

5.2.1.3 Forma de armazenamento

Os arquivos do RCS nada mais são do que arquivos. As revisões são armazenadas sob a forma de árvore, onde há um tronco principal e, às vezes, ramos (chamados de *branchs*).

Por medida de eficiência na utilização do espaço de armazenamento, o RCS utiliza o conceito de delta, ou seja, o que realmente se armazena são as diferenças entre as revisões. No RCS é utilizado o conceito de delta negativo, pois se armazena completamente a última revisão e os deltas

necessários para se recuperar alguma revisão mais antiga. Isto foi feito devido a uma pesquisa efetuada na Universidade de Purdue, onde se verificou que 90% das operações de retirada de item (*check-out*) eram realizadas sobre a última revisão.

Para o tratamento dos ramos (*branchs*), foi utilizado o conceito de delta positivo, pois neste caso não se guarda revisões completas, mas se monta a revisão a partir de seu ancestral no tronco raiz. No RCS a numeração referente ao arquivo consegue definir claramente o parentesco entre as revisões, o que não acontece no SCCS.

No RCS não é permitida a criação de deltas intermediários entre duas revisões , ou seja, se o último delta da revisão 1 é 1.5 e já existem deltas da revisão 2, não se consegue criar o delta 1.6, como no SCCS.

5.2.1.4 Sincronismo

O sincronismo de alterações é garantido através do uso do comando de *check-out* com *lock* (co -l); desta maneira, o arquivo estará bloqueado para outras alterações até que ele seja devolvido para o repositório. Isto garante que apenas uma pessoa por vez consiga alterar o mesmo arquivo. No entanto, o sistema permite que o bloqueio seja quebrado, mas uma mensagem é enviada para mostrar à pessoa que esta operação pode resultar em perda da integridade do repositório.

5.2.1.5 Tratamento de configurações

O RCS foi desenvolvido para tratar de alterações de itens individuais, mas permite que a um grupo de revisões seja dado um nome lógico que represente uma configuração. No entanto, a única operação que pode ser efetuada sobre o grupo como um todo é a recuperação por meio do comando de *check-out*.

5.2.1.6 Acesso aos dados

Para disciplinar o acesso aos arquivos sob guarda do RCS, pode-se criar uma lista de acesso (associada a cada arquivo) que contenha os nomes das pessoas com direito de operar sobre estes arquivos.

5.2.2 A Poke-Tool

A ferramenta Poke-Tool (**Po**tencial Uses **Cr**iteria **Tool** for Program Testing) é um ambiente de teste para análise de cobertura estrutural que apóia os critérios Potenciais-Usos[47], e outros critérios estruturais[17][47]. A versão aqui utilizada apóia o teste de unidade em programas C (entendendo por unidade uma função em C). No entanto, ela é uma ferramenta multilinguagem e pode trabalhar com programas escritos em Clipper[10], COBOL[46] e FORTRAN-77[25].

A Poke-tool é uma ferramenta orientada à sessão (que deve ser entendida como o conjunto das atividades de: fornecer o programa para teste, compilá-lo, gerar um programa instrumentado, executar os casos de teste, avaliar os casos de teste e visualizar os resultados) e é bastante modular. Na versão utilizada, a execução se dá por meio de *scripts* , mas também há uma versão com interface gráfica em padrão MOTIF[53] e uma proposta de extensão para apoiar o tratamento da não executabilidade[14].

Segundo o manual da ferramenta, seu uso básico está centrado em 4 passos. O primeiro executa a análise estática do programa em teste, o segundo compila o programa instrumentado no passo anterior, o terceiro executa os casos de teste e no quarto passo se avalia a cobertura atingida com os casos de teste executados. A chamada da ferramenta de teste é feita no diretório de teste (**TESTDIR**), o qual é criado na fase de integração do ambiente de GCST. O programa que será utilizado como exemplo é o programa **`prime.c`**, que recebe como entrada um número inteiro e identifica se o número é primo, ou não. A entrada 0 (zero) indica saída do programa.

5.2.2.1 Realizando a análise estática

O primeiro passo é executado com o comando **poketool**. Esse comando recebe como parâmetros o nome do arquivo do programa em teste e também o nome da função que será devidamente testada. Com sua execução obtém-se informações descritivas do programa, preparando-o, também, para a fase de análise dinâmica, que será realizada posteriormente, a qual consiste na execução dos casos de teste. Veja na Figura 5 a chamada da ferramenta para o programa **`prime.c`** onde será testada a função **main**.

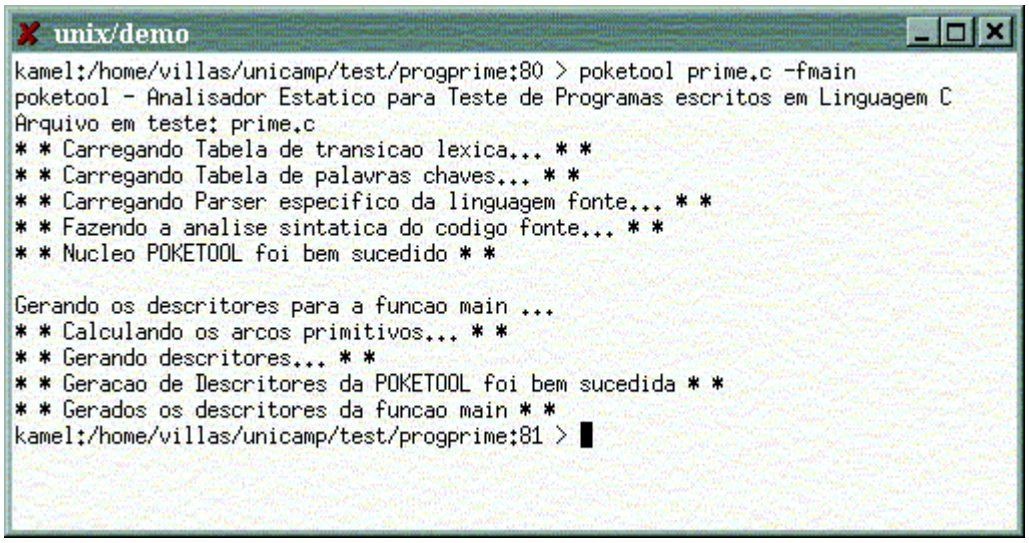

Figura 5 - Chamada da **Poke-tool**

Com a execução da poke-tool, o programa em teste (**`prime.c`**) é lido, é executada a análise estática do mesmo e são gerados o programa instrumentado para teste (**`testeprog.c`**) e as informações descritivas. O programa instrumentado está localizado no diretório de teste, já as informações descritivas estão localizadas num subdiretório de mesmo nome da função em teste (**/main**) criado abaixo do diretório de teste.

O programa instrumentado é um programa funcionalmente equivalente ao programa original (**`prime.c`**), mas com comandos de escrita em arquivo (chamados de pontas de prova) os quais permitem que os caminhos executados pelos casos de teste sejam obtidos. Veja na Figura 6 uma parte do programa instrumentado, observe que os números à esquerda indicam o número do bloco a que pertencem os comandos vistos no lado direito.

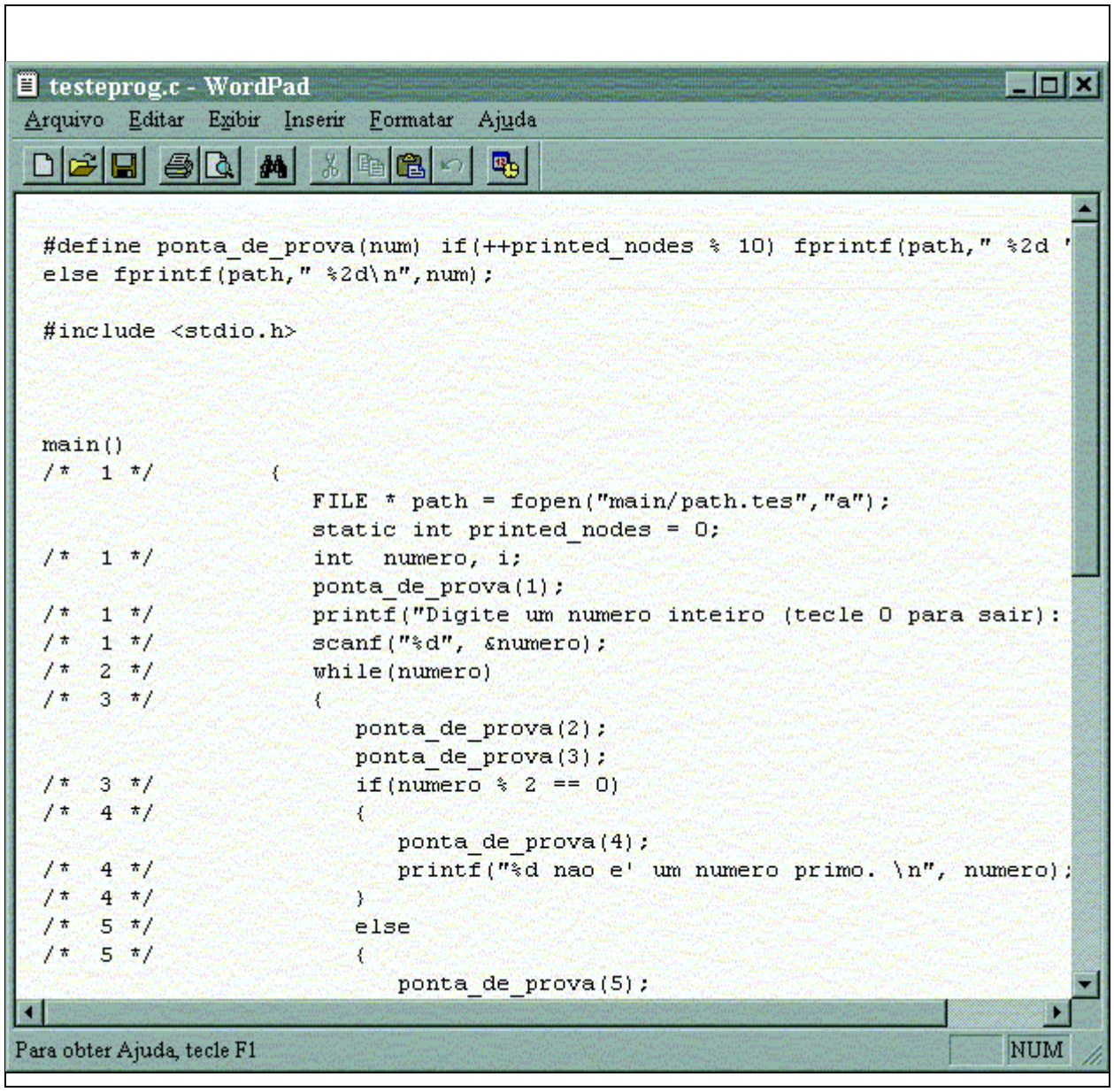

Figura 6 - O programa instrumentado **`testeprog.c`**

5.2.2.2 Gerando o programa executável para teste

Para a execução dos casos de teste o programa **`testeprog.c`** deve ser compilado. A compilação é feita com o comando **gcc testeprog.c –o testeprog**. Dessa forma é gerado o programa executável **`testeprog`**, que será usado no passo seguinte.

5.2.2.3 Executando os casos de teste

O terceiro passo é executado com o comando **pokeexec**. Esse comando recebe como parâmetros o nome do programa executável, o tipo de entrada de dados (se por parâmetro ou pelo teclado), o nome da função que se está testando. Ele aciona o programa em teste, e dessa forma, pode-se entrar com os dados de teste e obter os resultados. É necessário que, após a execução de cada caso de teste, se informe à ferramenta se outro caso de teste será, ou não, inserido. Veja sua execução na Figura 7.

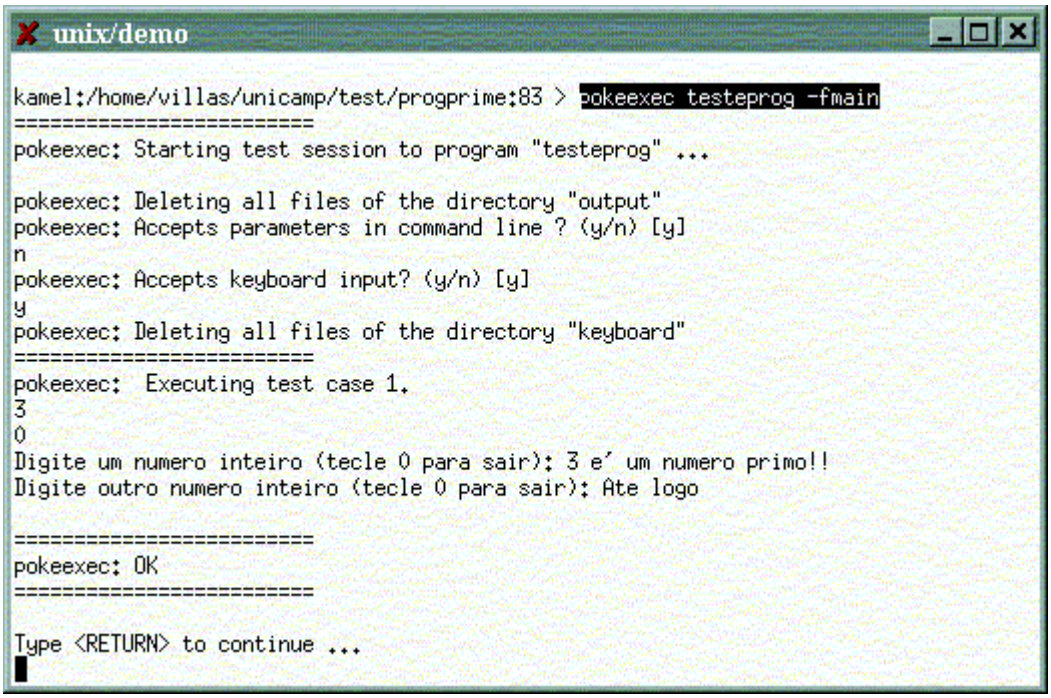

Figura 7 - Execução do comando **pokeexec**

É nesse ponto do processo de teste que o testador examina se a saída obtida coincide com a saída esperada, a qual é definida pela especificação do programa em teste. Caso as saídas não estejam em conformidade, o defeito deve ser corrigido e o teste deve recomeçar pelo primeiro passo.

Com a execução do comando **pokeexec** e dos casos de teste, são criados dois subdiretórios localizados abaixo do diretório de teste, **/keyboard** e **/output**. No primeiro são encontradas informações sobre as entradas fornecidas e no segundo informações sobre as saídas obtidas. No subdiretório da função em teste (**/main**) agora podem ser encontrados os caminhos que cada caso de teste produziu armazenados no arquivo **path.tes**.

5.2.2.4 Avaliando a cobertura atingida

O quarto passo é executado com o comando **pokeaval**. Ele recebe como parâmetros o nome da função que se está testando, o critério de cobertura escolhido, a forma de apresentação dos elementos requeridos pelo critério escolhido e o intervalo de casos de teste que se quer avaliar. Esse comando utiliza os elementos requeridos gerados pelo comando **poketool** e os caminhos executados pelos casos de teste, os quais foram gerados pelo comando **pokeexec**. Desse modo, ele é capaz de calcular e informar quais elementos requeridos foram exercitados, ou não. Como saída esse comando apresenta os elementos requeridos ainda não exercitados e a cobertura atingida. Veja seu uso na Figura 8.

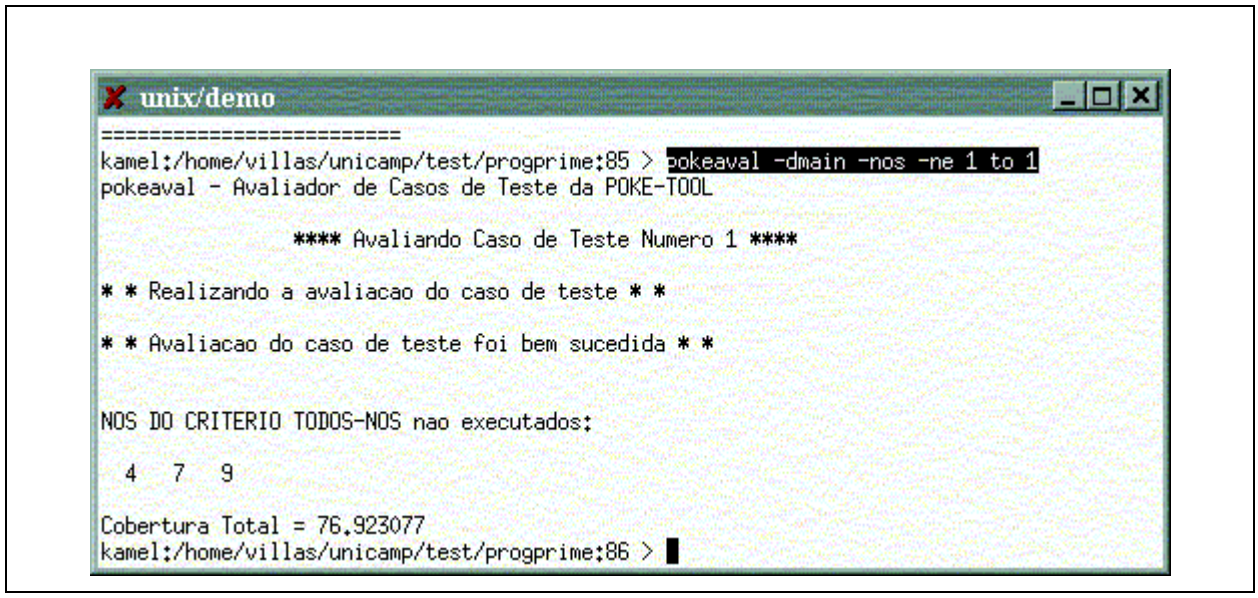

Figura 8 - Execução do comando **pokeaval**

5.3 A implementação do ambiente de gestão de configuração para teste

O ambiente de apoio à gestão de configuração de software para teste (GCST) é implementado por meio de uma estrutura de diretórios com repositórios controlados RCS[63] e um conjunto de comandos que manipulam a informação coletada durante o processo de teste. Esses comandos foram implementados por meio de um conjunto de *shell scripts* que encapsulam os comandos básicos do RCS.

Foram implementados 6 comandos com a seguinte funcionalidade:

1. **qconfigura:** comando interativo para inserção ou remoção de ferramentas e seus respectivos critérios associados. Cria ou remove a estrutura de diretórios necessária para cada ferramenta abaixo do diretório de configuração (**conf**), como visto na Figura 3.

2. **qcriast:** comando para o cadastro de uma nova sessão de teste. Caso a ST não exista, cria o diretório da ST abaixo do diretório **LIBDIR** e abaixo dele o diretório de trabalho **WORK** e o repositório RCS. No diretório criado para a ST, cria o arquivo **ArqST** com as informações do responsável pelo teste, nome da ferramenta de teste, nome do critério associado. Caso haja documentos de teste, ele também cria o arquivo **ConfST** com as informações dos documentos (arquivo, endereço, descrição). A Figura 4 ilustra esse caso.

3. **qcriasac:** comando para cadastro das sessões de análise de cobertura. Caso a SAC não exista para aquela ST é criado no diretório da ST o arquivo **ArqSAC** com as informações do critério e da ferramenta utilizados para aquela SAC.

4. **qsalvasac:** comando para salvamento intermediário das informações de uma SAC. Caso a SAC e ST informadas sejam válidas, obtém as informações necessárias sobre o ICT relativo no arquivo **ArqIC**, apontado pelo nome da ferramenta e do critério apoiado, no diretório de configuração **conf**. Com essas informações, faz acesso aos arquivos componentes do ICT e os salva no repositório RCS da ST, com o motivo do salvamento.

5. **qcongelasac:** comando para criação de uma configuração-base de uma SAC. Caso a SAC e ST informadas sejam válidas, obtém as informações necessárias sobre o ICT relativo no arquivo **ArqIC**, apontado pelo nome da ferramenta e do critério apoiado, no diretório de configuração **conf**. Com essas informações, faz acesso aos arquivos componentes do ICT, salva-os no repositório RCS da ST e cria a configuração-base da SAC, protegendo esses dados contra escrita.

6. **qrecuperasac:** comando para recuperação de uma SAC. Caso a SAC e ST informadas sejam válidas, obtém as informações necessárias sobre o ICT relativo no arquivo **ArqIC**, apontado pelo nome da ferramenta e do critério apoiado, no diretório de configuração **conf**. Com o nome da SAC recupera a configuração-base do repositório RCS da ST e a coloca no diretório **WORK**. Com as informações de nome e endereço dos arquivos componentes do ICT obtidas no arquivo **ArqIC**, os dados do ICT são colocados no diretório da ferramenta de teste para utilização posterior.

5.4 Exemplo prático de uso da ferramenta de GCST

Vamos, agora, mostrar a atuação de um testador no ambiente proposto. Será descrita a caracterização do item de configuração de teste para a ferramenta de teste em uso, a configuração do ambiente de controle para integrar a ferramenta de teste e o uso dos comandos desse ambiente para gestão das informações geradas ao longo da atividade de teste.

5.4.1 Caracterizando o item de configuração de teste

De acordo com o exposto no capítulo anterior, o ICT do exemplo é composto por arquivos gerais:

- prime.c (programa fonte em teste),
- testeprog.c (programa instrumentado para teste),
- main.gcf (grafo de fluxo de controle),
- grafodef.tes (grafo de fluxo de dados),
- keyboard<n>.tes (dados de teste),
- output<n>.tes (saídas reais),
- path<n>.tes (caminho percorrido),
- Plan Tes do prime.doc (Plano de teste para o programa prime.c);

por arquivos relacionados ao critério potenciais usos:

- exec_pu.tes (associações executadas),
- puoutput.tes (associações não executadas),
- puhis.tes (referência às associações requeridas exercitadas);

por arquivos relacionados ao critério todos os arcos:

- exec_arc.tes (arcos executados),
- arcoutput.tes (arcos não executados),
- archis.tes (referência aos arcos requeridos exercitados);

e por arquivos relacionados ao critério todos os nós:

- exec_no.tes (nós executados),
- nosoutput.tes (nós não executados),
- noshis.tes (referência aos elementos requeridos exercitados).

5.4.2 Preparando o ambiente de software

A primeira atividade é preparar o ambiente de software, criando os diretórios de trabalho e declarando as variáveis de ambiente UNIX necessárias para a operação do ambiente de gestão de configuração para teste. Para isso, as seguintes tarefas devem ser executadas:

1. Criar os diretórios de trabalho (que no exemplo são):
   - `mkdir /home/villas/unicamp/test` (diretório de teste)
   - `mkdir /home/villas/unicamp/test/conf` (diretório de configuração da ferramenta de gestão de configuração de teste)
   - `mkdir /home/villas/unicamp/test/lib` (diretório das sessões de teste)
   - `mkdir /home/villas/unicamp/test/andre/bin` (diretório em que os *scripts* do ambiente de GCST serão instalados)
2. Declarar o caminho dos *scripts* no arquivo ***.user_login***:
   - setenv PATH ${PATH}:/home/villas/unicamp/test/andre/bin
3. Declarar as seguintes variáveis de ambiente no arquivo ***.user_cshrc***:
   - setenv TESTDIR /home/villas/unicamp/test
   - setenv CONFDIR /home/villas/unicamp/test/conf
   - setenv LIBDIR /home/villas/unicamp/test/lib

5.4.3 Integrando a ferramenta de teste ao ambiente de GCST

Após a preparação do ambiente de software, a integração da ferramenta de teste Poke-tool ao ambiente de GSCT é feito com o comando **qconfigura** do modo como está mostrado na Figura 9 a seguir. Esse comando é interativo e por meio de perguntas feitas ao testador, obtém as informações sobre as ferramentas de teste, os critérios apoiados e sobre os arquivos componentes dos vários ICT (nome do arquivo, endereço do arquivo, descrição).

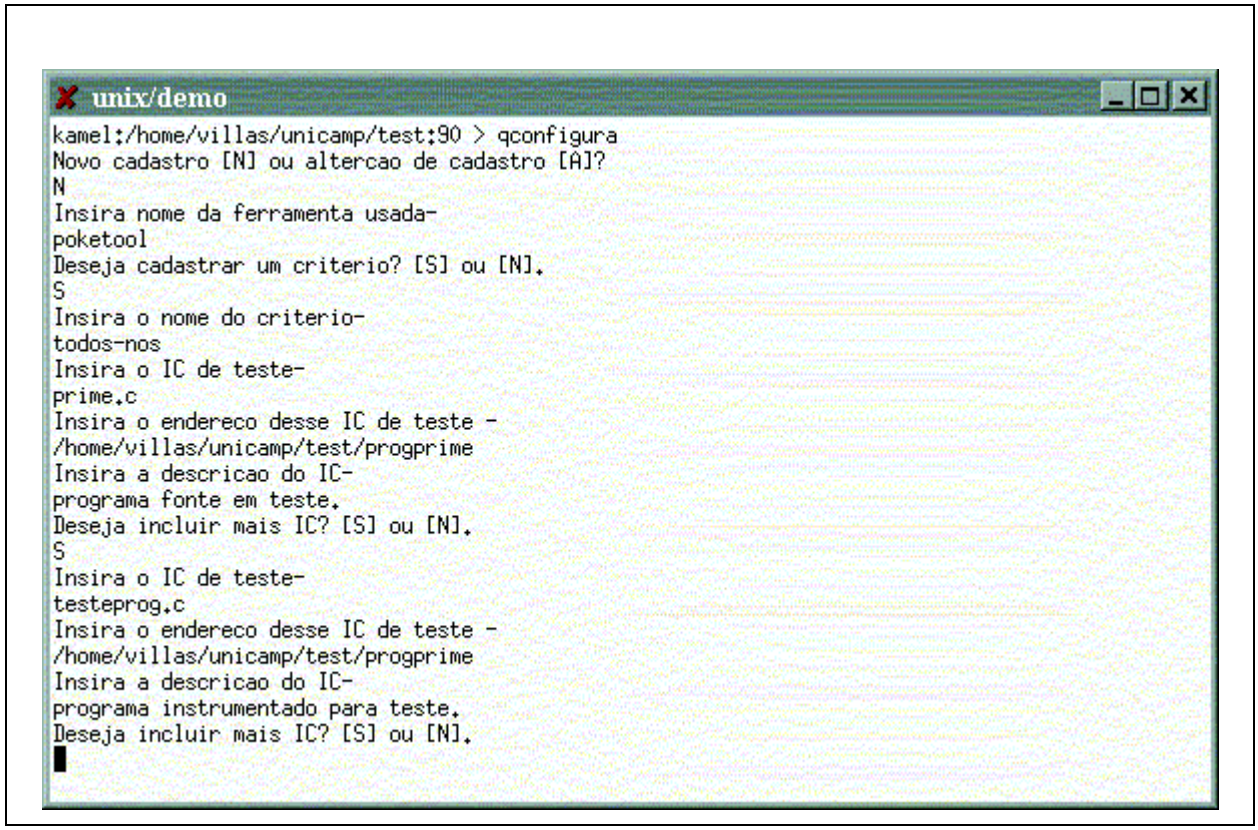

Figura 9 - Uso do comando **qconfigura**

Observe no diretório mostrado na Figura 10, que a ferramenta Proteum também está integrada, mas, como ela não será usada, ele a remove usando o mesmo comando **qconfigura** da forma mostrada na Figura 11.

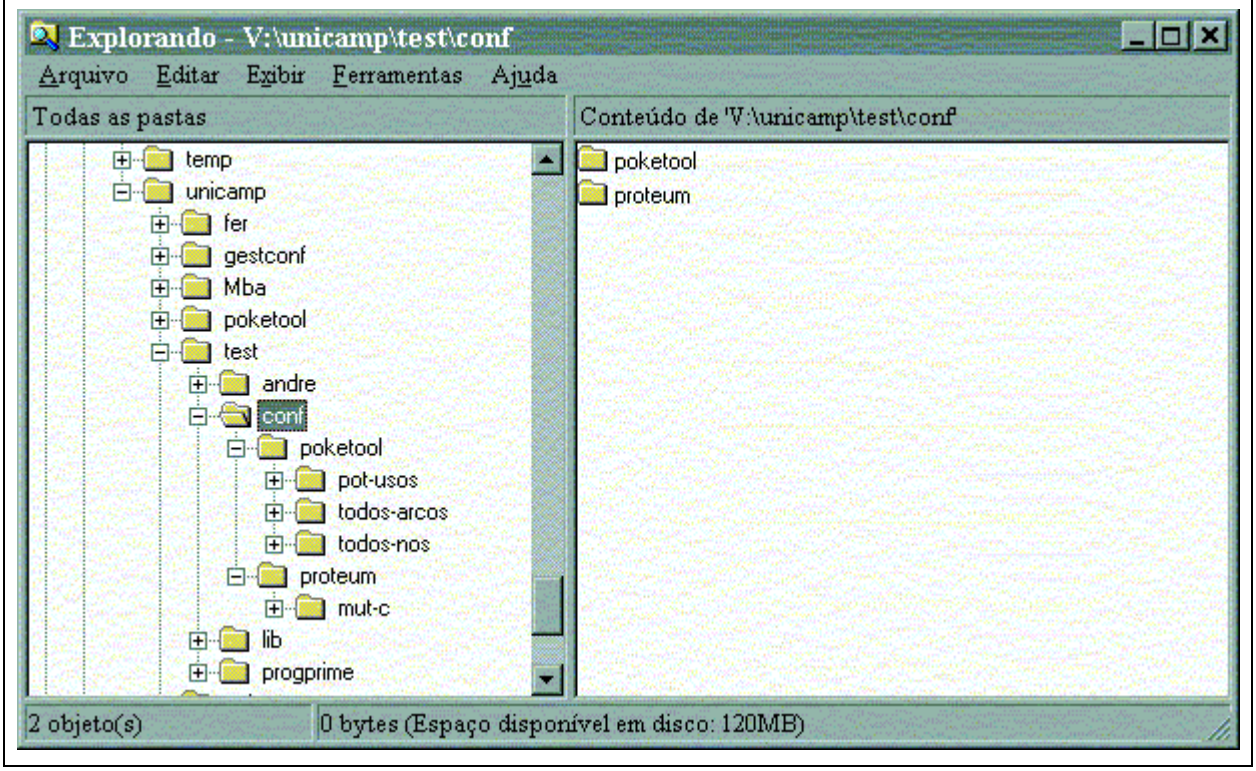

Figura 10 - Diretório de configuração com as ferramentas de teste

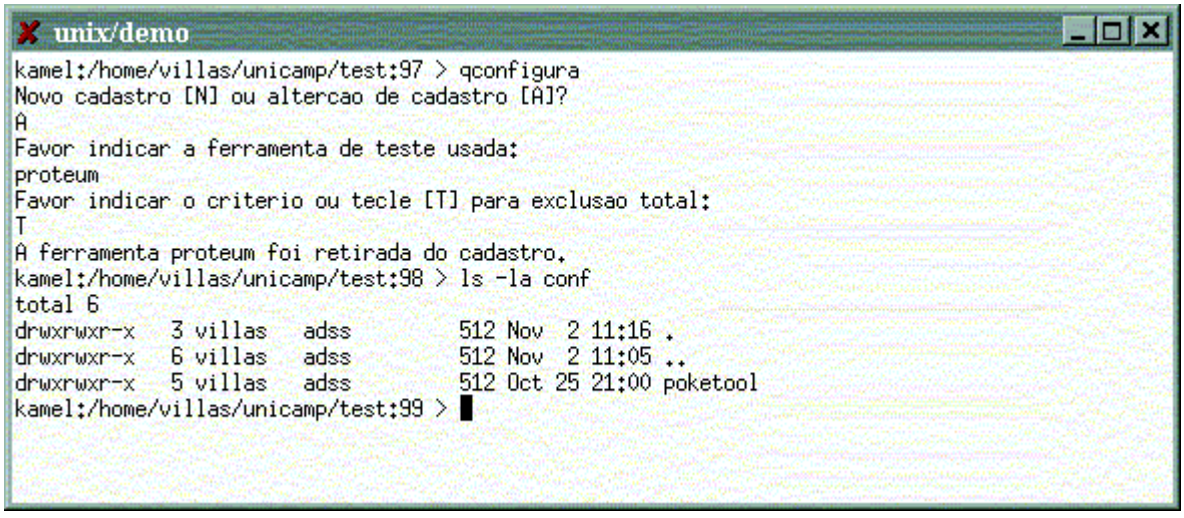

Figura 11 - Diretório de configuração após retirada da Proteum

Nos subdiretórios correspondentes a cada critério utilizado, foi criado, para cada um deles, um arquivo de configuração do critério (veja **ArqIC** na Figura 12), cujo conteúdo pode ser visto, em

parte, na Figura 13.

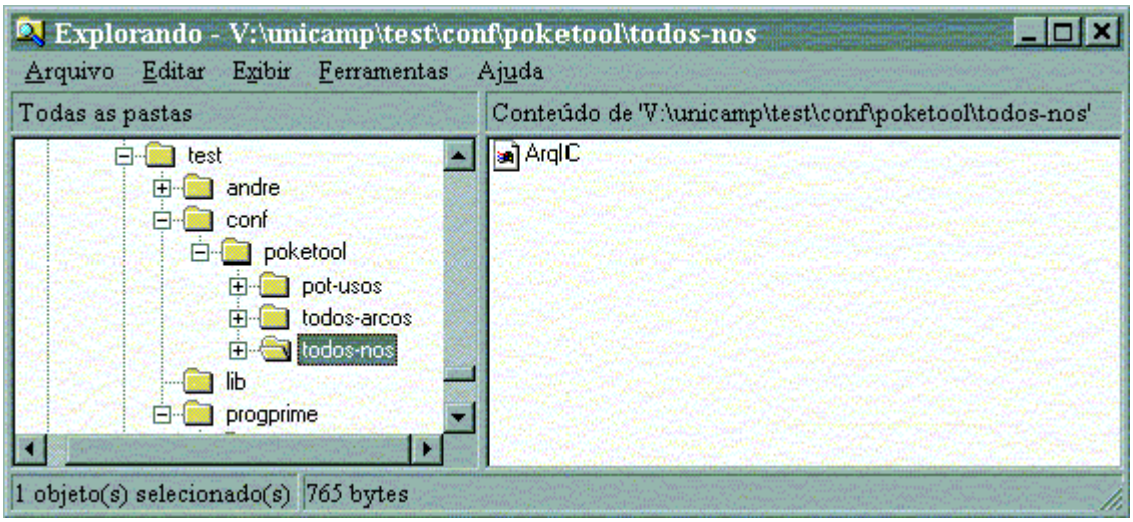

Figura 12 - Diretório do critério todos os nós no diretório de configuração

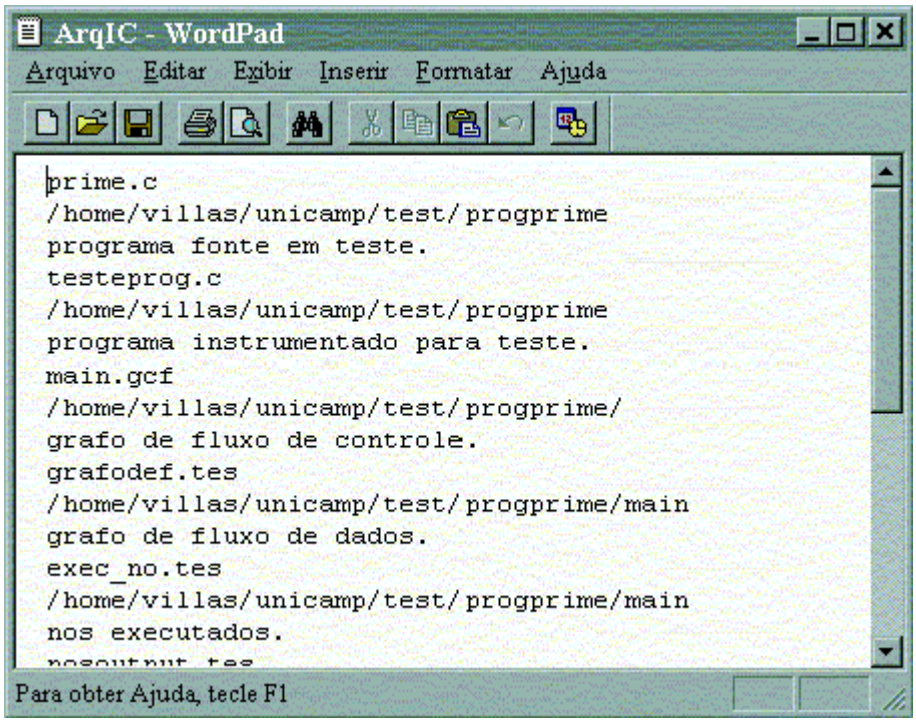

Figura 13 - Conteúdo do arquivo **ArqIC** para o critério todos os nós

5.4.4 Criando uma sessão de teste

Com a ferramenta configurada para entender os dados da Poketool, ele passa a criar o espaço de controle de sua sessão de teste, que é uma estrutura de diretório que fica abaixo do diretório **LIBDIR**, já declarado e que, como pode ser visto na Figura 14, está vazio.

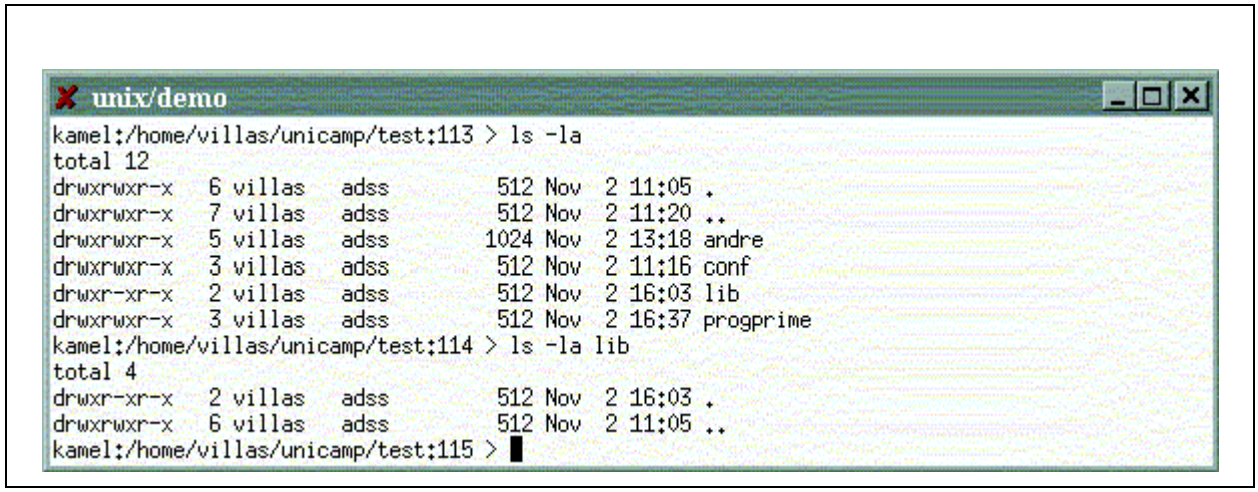

Figura 14 - Diretório das sessões de teste

Com o comando **qcriast**, ele vai criar uma sessão de teste de nome **st1**, que aceita a poketool como ferramenta de teste apoiando os critérios: todos os nós, todos os arcos e potenciais usos, como está apresentado na Figura 15.

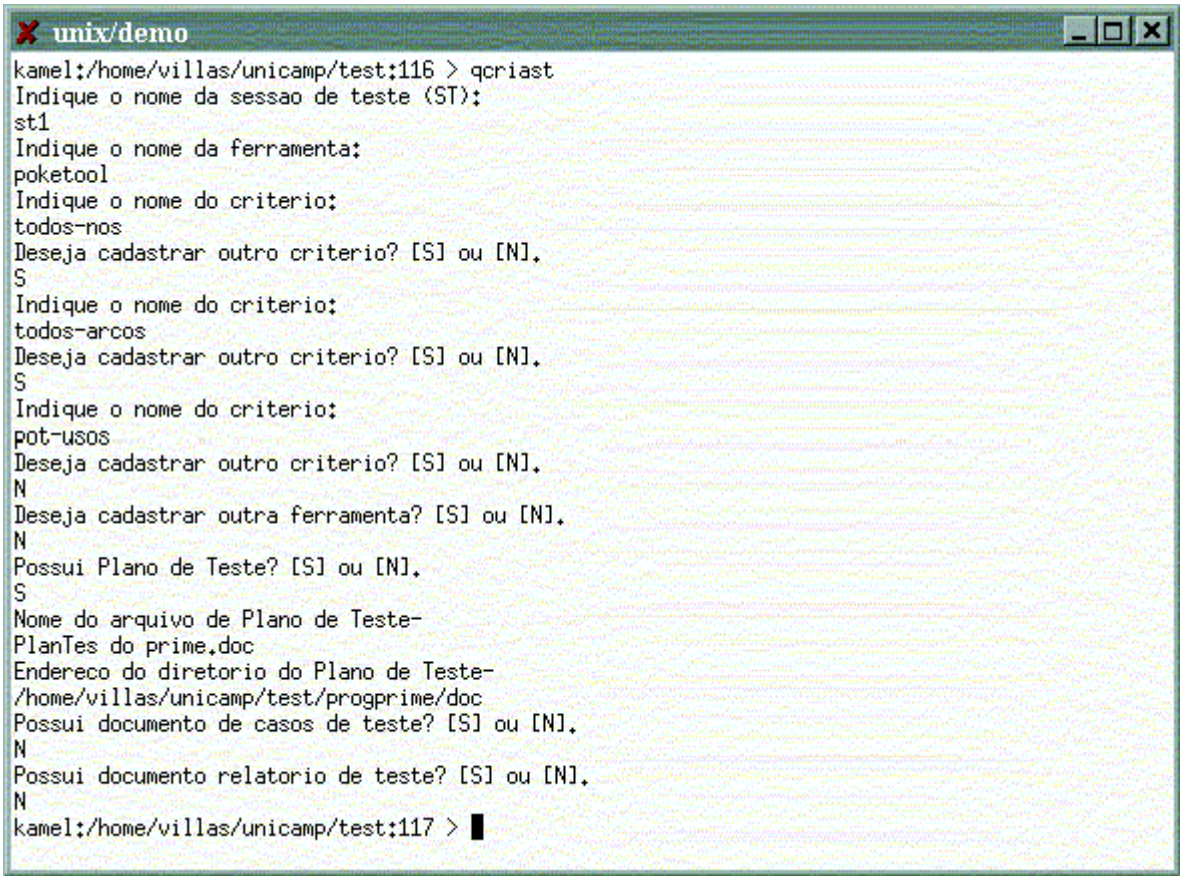

Figura 15 - Uso do comando **qcriast**

Veja na Figura 16 como está o diretório para a sessão de teste recém criada (**st1**).

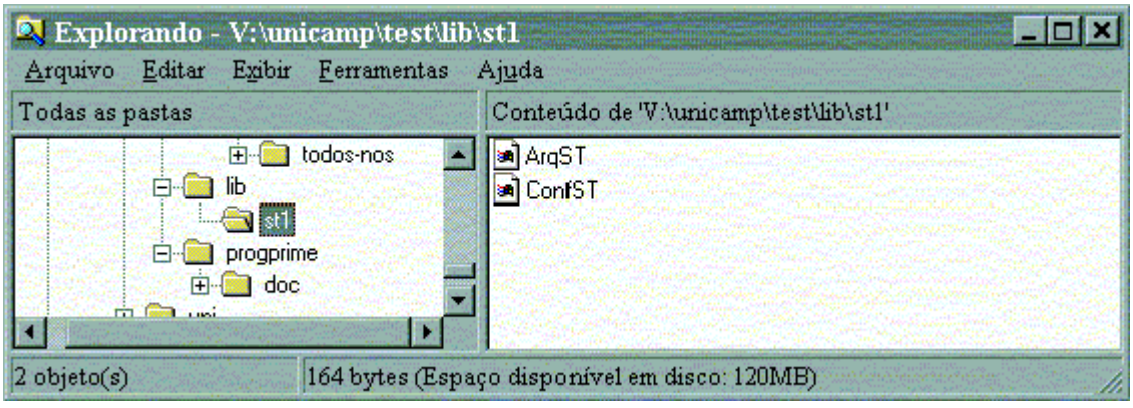

Figura 16 - Diretório completo para sessão de teste **st1**

Como explicado anteriormente, os arquivos **ConfST** e **ArqST** criados contêm, respectivamente, o nome, endereço e descrição do documento de planejamento dos testes, e informações sobre a sessão de teste (nome do usuário, nome da ferramenta e nome dos critérios apoiados pela ferramenta). Veja Figura 17 e Figura 18.

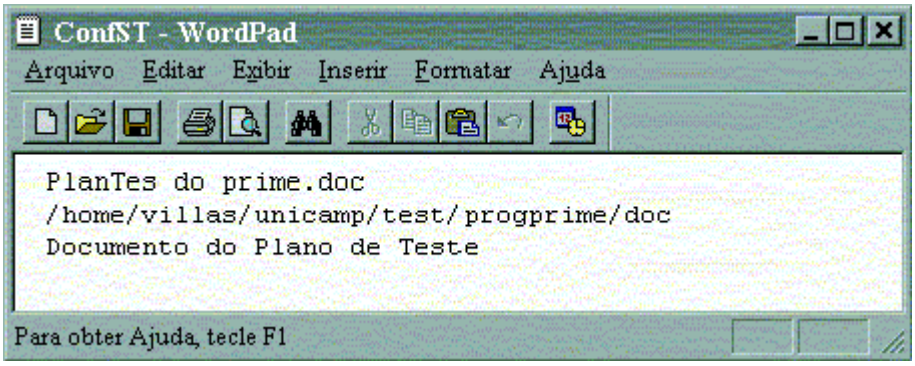

Figura 17 - Conteúdo do arquivo **ConfST**

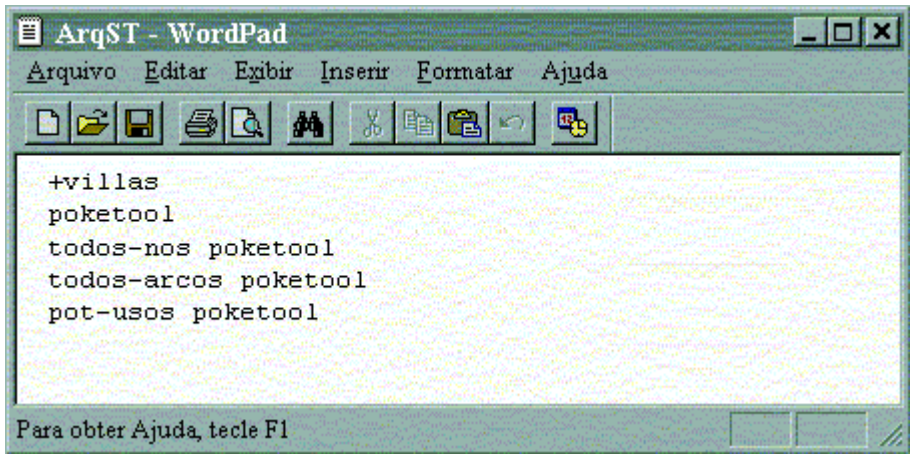

Figura 18 - Conteúdo do arquivo **ArqST**

5.4.5 Executando os testes

A execução dos testes é iniciada nesse ponto, quando o testador executa a ferramenta de teste poketool para o programa em questão (`prime.c`), a qual gera um programa instrumentado para teste (`testeprog.c`) que deve, logo a seguir, ser compilado para poder ser executado com os casos de teste escolhidos. A seguinte seqüência de comandos deve ser executada, utilizando a ferramenta poketool:

>poketool prime.c –fmain (executa a poketool para o programa prime.c),

>gcc testeprog.c –o testeprog (compila o programa testeprog)

Observe na Figura 19 que, no diretório **progprime,** estão o programa em teste (`prime.c`) e o diretório onde se encontram os documentos relativos à atividade de teste (diretório **doc**), no caso, o plano de teste.

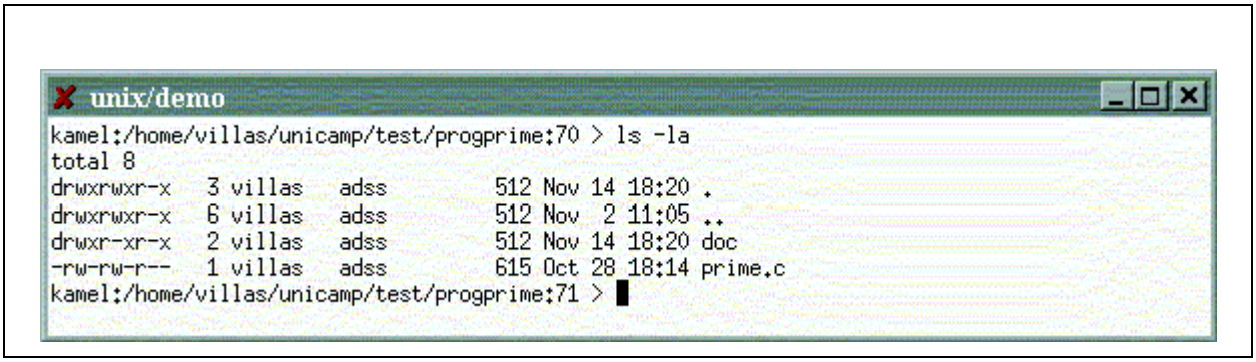

Figura 19 - Diretório **progprime** antes de executar a poketool

Após a execução da ferramenta de teste (poketool) conforme demonstrado acima, são criados o programa intrumentado para teste (testeprog.c) e o diretório **main**, que contém uma série de informações sobre a análise estática do programa `prime.c`, como pode ser visto na Figura 20. Compilando o programa testeprog.c, pode-se começar a executar o programa para fins de teste.

5.4.6 Criando uma SAC para o critério todos-os-nós

Preparado para executar o programa em teste, o testador cria o ambiente para proteção dos dados de sua primeira sessão de análise de cobertura, que, segundo o plano de teste, será realizada utilizando o critério todos-os-nós. A sessão de análise de cobertura é criada com o comando **qcriasac**, recebendo como parâmetros o nome da SAC, o nome da ST, o nome da ferramenta de teste e o critério utilizado.

>qcriasac sac1 st1 poketool todos-nos

Nesse momento é criado no diretório da sessão de teste (st1) o arquivo relativo à SAC (**ArqSAC**) e o diretório de trabalho (**WORK**) que contém o repositório RCS. Veja isso na Figura 21 e compare com a Figura 16, para ver a diferença.

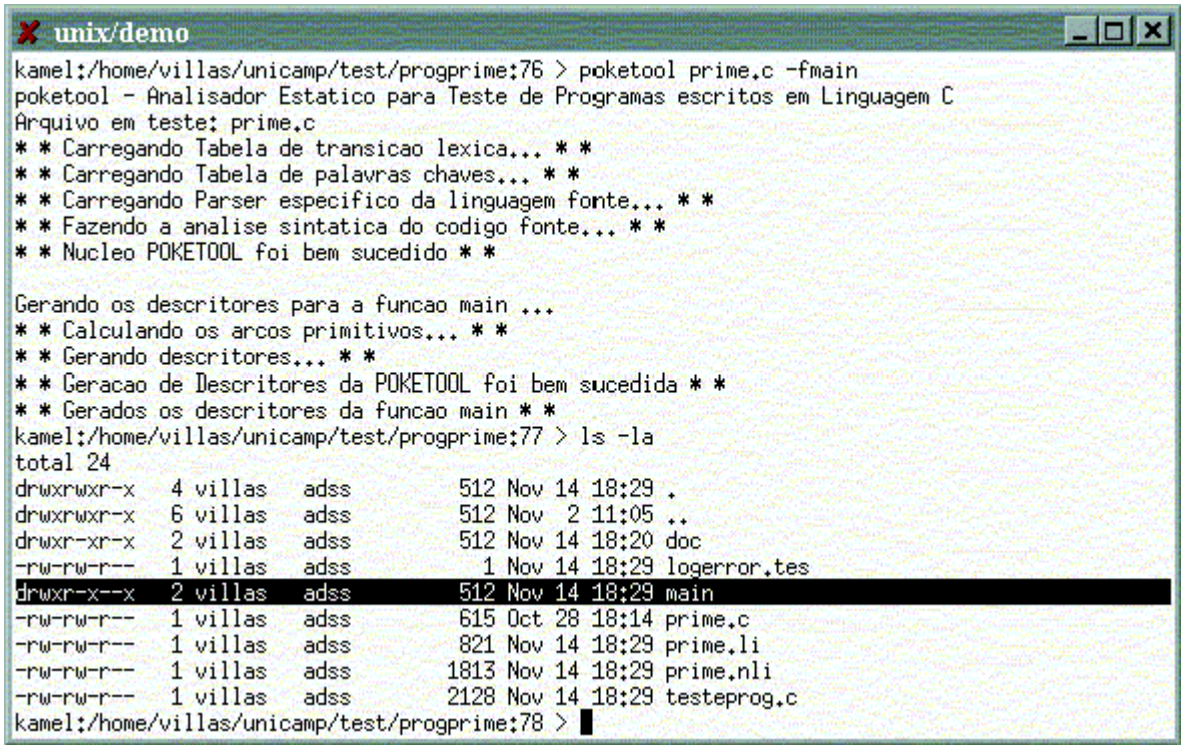

Figura 20 - Diretório **progprime** após execução da poketool

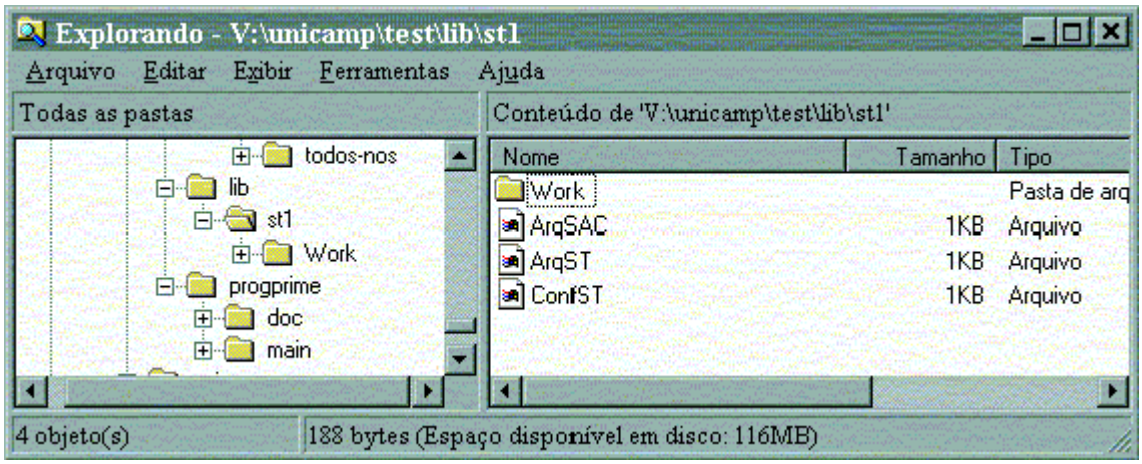

Figura 21 - Diretório da **st1** após criação da **sac1**

A poketool deve, então, ser executada para o critério todos-os-nós com o caso de teste previsto no plano de teste (CT[3,0]). E, logo após, deve ser verificada a cobertura obtida com o caso de teste referido. Os seguintes comandos executam a tarefa descrita.

>pokeexec testeprog –fmain (executa o programa instrumentado com os dados de teste escolhidos)

>pokeaval –dmain –nos –ne 1 to 1 (avalia a cobertura obtida para o critério todos os nós com o dado de teste utilizado)

5.4.7 Salvamento intermediário de dados de teste

Observe na Figura 22 que a cobertura não atingiu o nível desejado. No entanto, por motivos diversos, o testador pode querer salvar esse resultado e a ferramenta possibilita essa tarefa. Isso é feito com o comando **qsalvasac**, recebendo como parâmetros o nome da SAC, o nome da ST e o motivo desse salvamento intermediário.

>qsalvasac sac1 st1 "sai mais cedo"

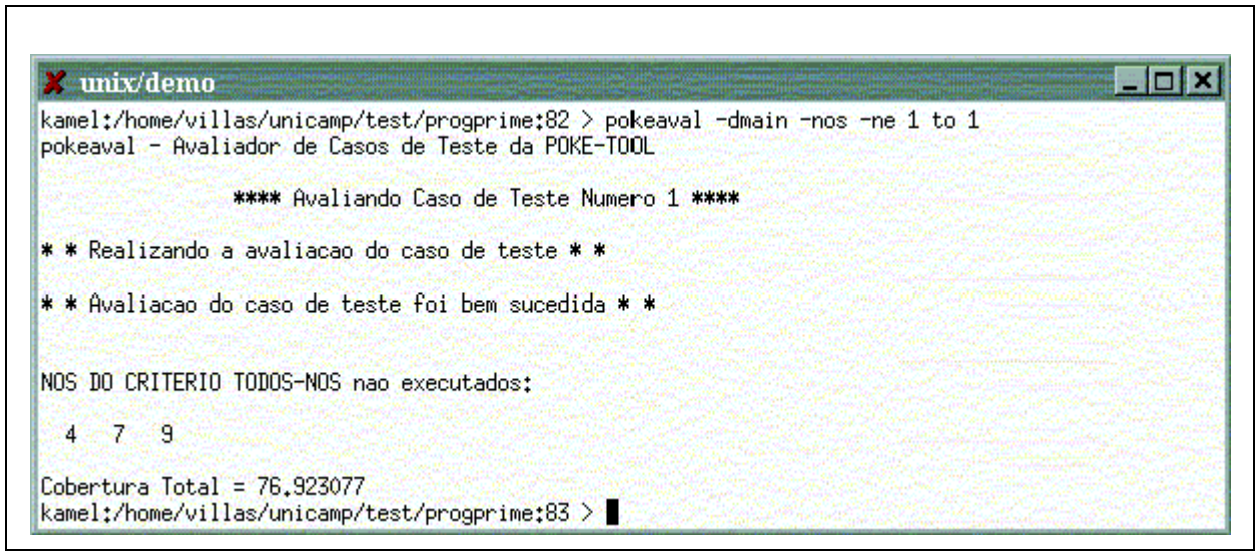

Figura 22 - Resultado da avaliação do primeiro caso de teste para a **sac1**

Observe na Figura 23 que com a execução desse comando o repositório RCS da sessão de teste **st1** agora contém os ICT com os 10 elementos definidos para o critério em questão no ato da configuração e que estão associados à **sac1**. A letra "v" após o nome do arquivo indica que ele é um arquivo do RCS. Apenas a título de esclarecimento, pode-se ver na Figura 24 o arquivo **path1.tes** no repositório RCS indicando a sua versão (0.0) e o motivo do salvamento ("sai mais cedo") no campo *log*.

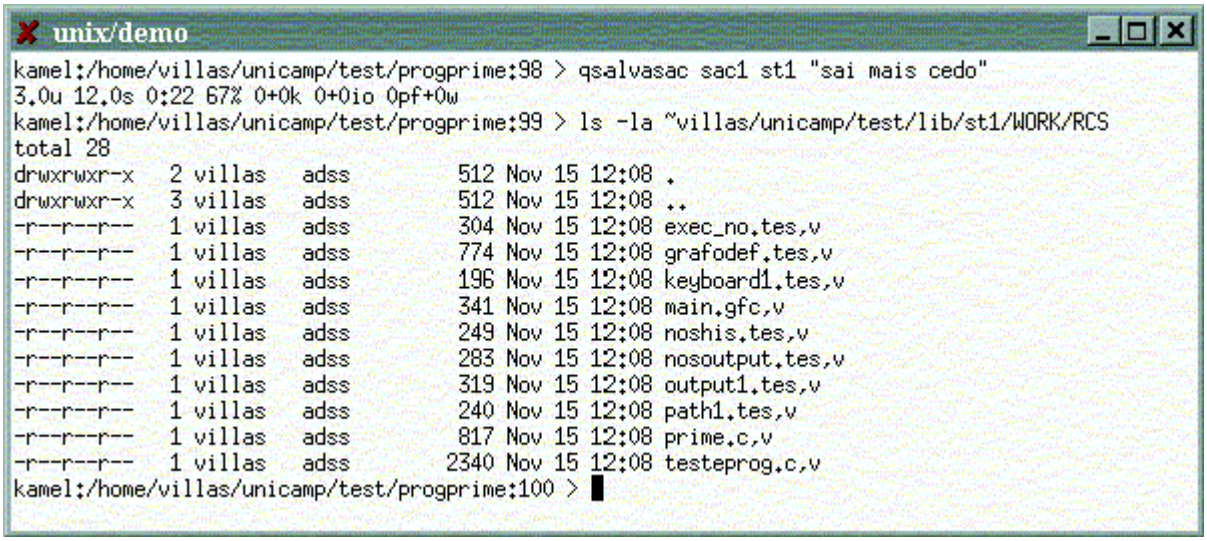

Figura 23 - Diretório RCS da **st1** após interrupção da **sac1**

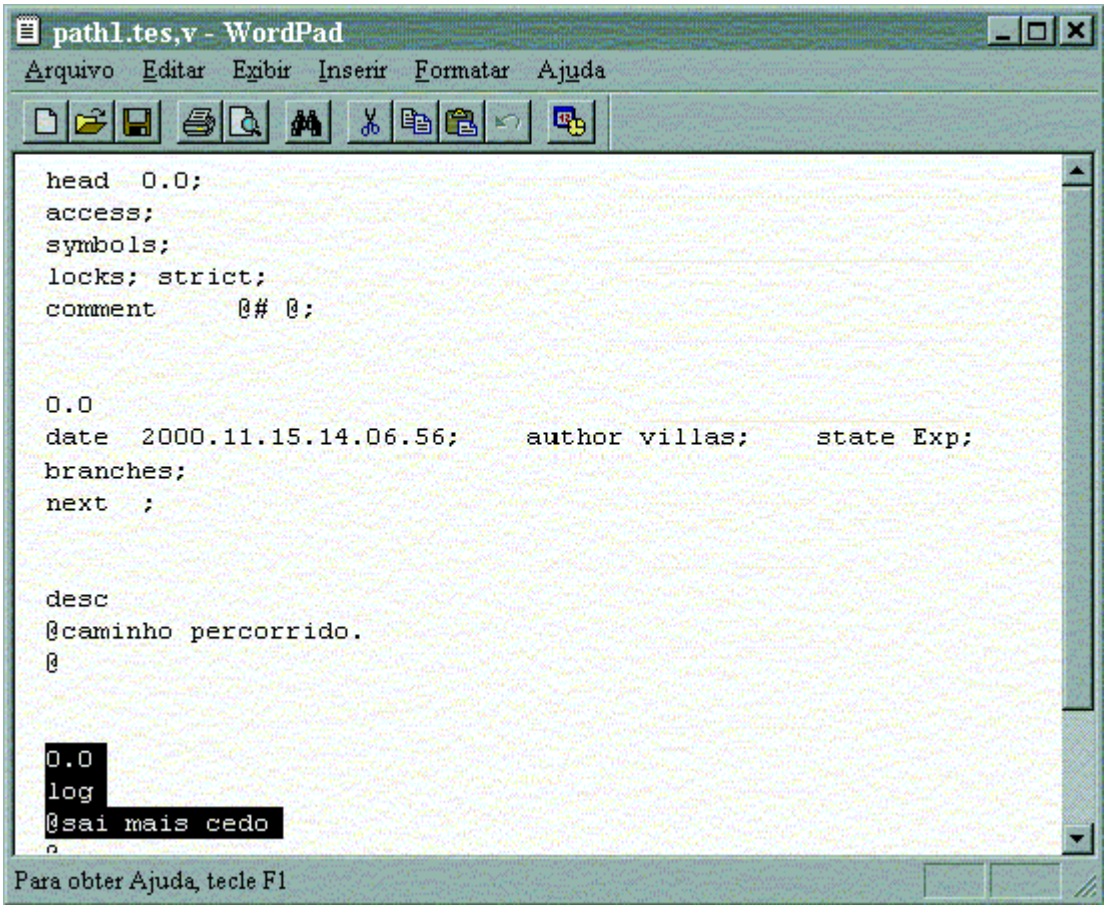

Figura 24 - Arquivo **path1.tes** no repositório RCS após interrupção da **sac1**

O teste, então, continua com a execução do caso de teste que garante a cobertura total para o critério todos os nós.

>pokeexec testeprog –fmain (executa o programa instrumentado com os dados de teste escolhidos)

>pokeaval –dmain –nos –ne 1 to 1 (avalia a cobertura obtida para o critério todos os nós com o dado de teste utilizado)

5.4.8 Criando uma configuração-base para o critério todos-os-nós

Observe na Figura 25 que, para um segundo caso de teste executado, a cobertura foi total (100%); dessa forma, de acordo com o planejado, essa SAC pode ser encerrada e seus dados devem ser protegidos.

O testador, então, congela esta SAC, criando a sua configuração-base (*baseline*), o que é feito com o uso do comando **qcongelasac** tendo como parâmetros o nome da SAC e o nome da ST em questão.

>qcongelasac sac1 st1

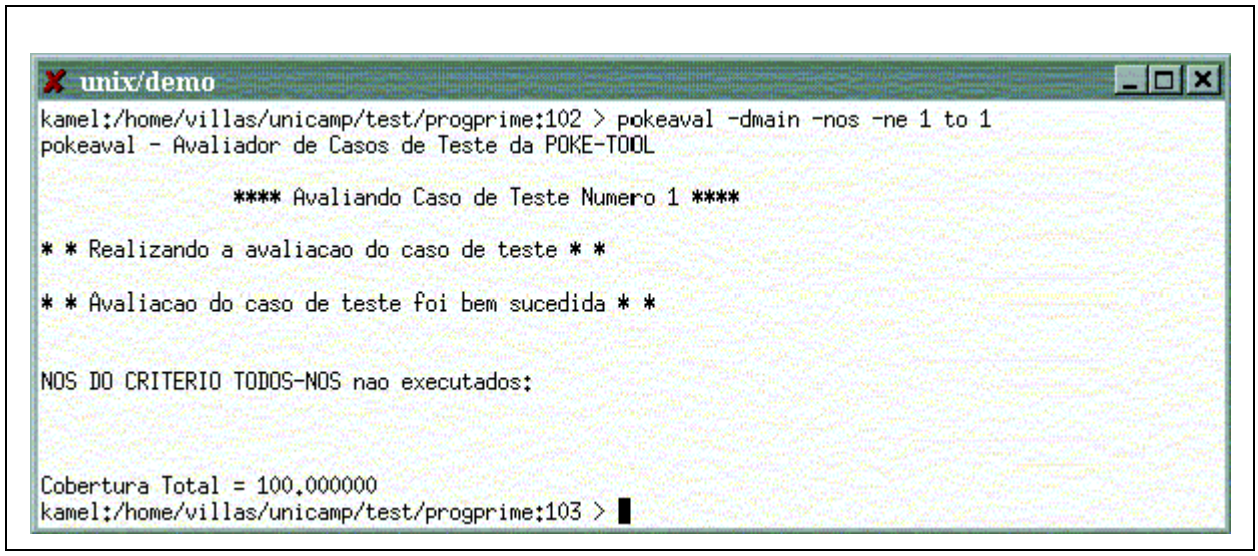

Figura 25 - Execução da avaliação do segundo caso de teste para a **sac1**

Esse comando vai colocar uma marca nos arquivos que estão no diretório RCS da st1, como pode ser visto na Figura 26, em que no campo *symbols*, aparece "SACbase-sac1:0.1;" indicando que a

versão 0.1 é a versão que foi congelada (*baselined*) com o nome lógico **SACbase-sac1**, ou seja, uma configuração-base da sac1. Compare com a Figura 24 e perceba que agora o campo *log* para a versão 0.1, que é o motivo do salvamento, diz "baseline da SAC sac1".

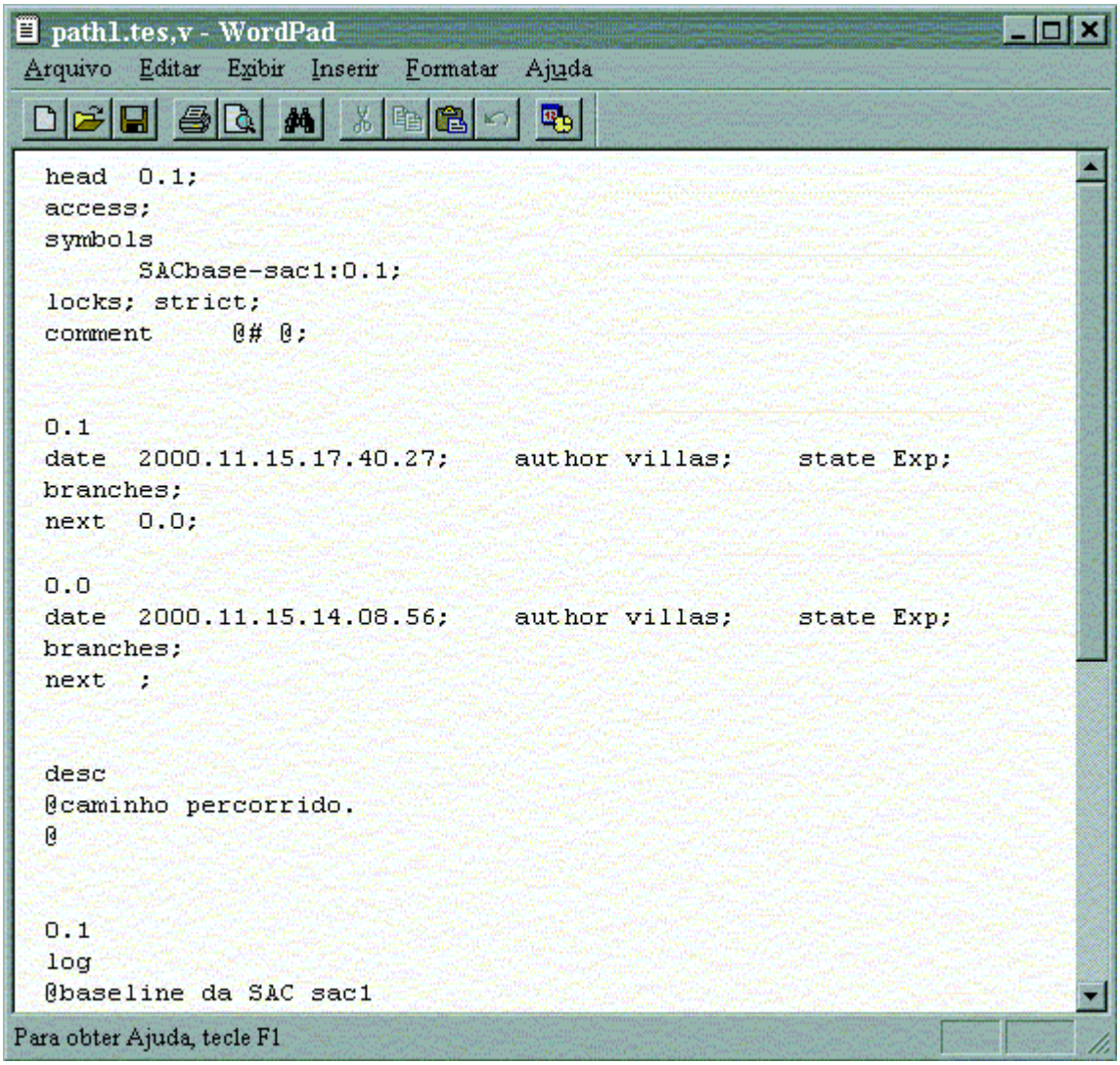

Figura 26 - Arquivo **path1.tes** no repositório RCS após congelamento da **sac1**

5.4.9 Criando uma SAC para o critério todos-os-arcos

Dando seqüência ao teste, o testador cria o ambiente para proteção dos dados de sua segunda sessão de análise de cobertura, que, de acordo com o plano de teste, será realizada utilizando o critério todos-os-arcos. A sessão de análise de cobertura, como já foi visto, é criada com o comando qcriasac.

>qcriasac sac2 st1 poketool todos-arcos

A poketool deve, então, ser executada para o critério todos os arcos com o caso de teste previsto no plano de teste.

>pokeexec testeprog –fmain (executa o programa instrumentado com os dados de teste escolhidos)

>pokeaval –dmain –arcs –ne 1 to 1 (avalia a cobertura obtida para o critério todos os arcos com o dado de teste utilizado)

5.4.10 Criando uma configuração-base para o critério todos-os-arcos

Observe na Figura 27 que a cobertura foi total (100%); dessa forma, o testador decide congelar esta SAC, criando a sua configuração-base (*baseline*), o que, como já foi mostrado, é feito com o uso do comando **qcongelasac**, tendo como parâmetros o nome da SAC e o nome da ST em questão.

>qcongelasac sac2 st1

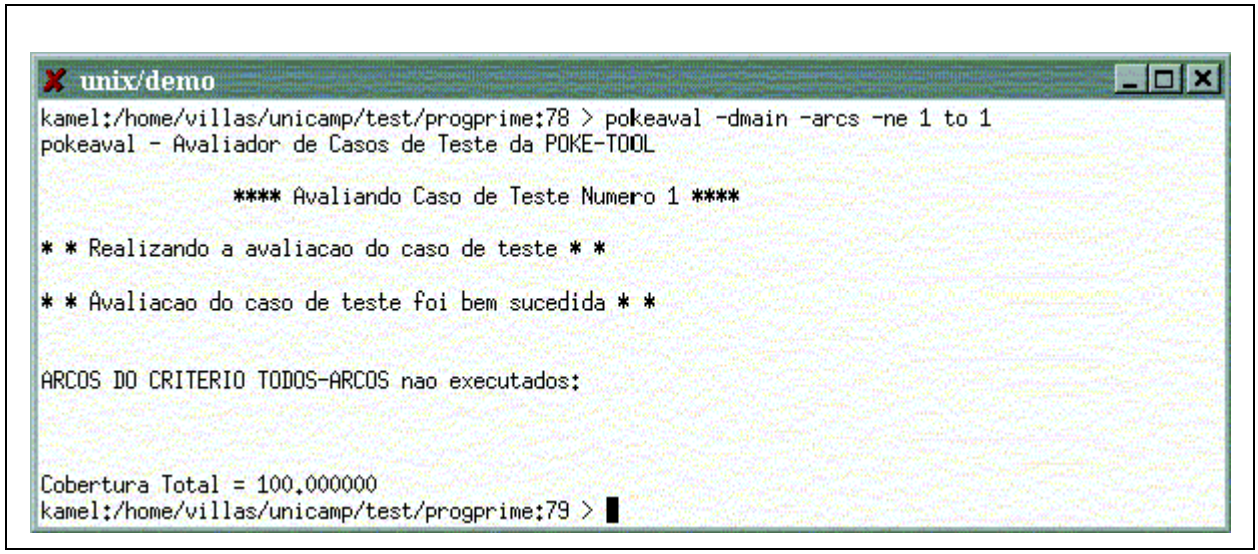

Figura 27 - Execução da avaliação do caso de teste para a **sac2**

5.4.11 Criando uma SAC para o critério potenciais-usos

Para finalizar o previsto no plano de teste dessa sessão de teste, uma terceira sessão de análise de cobertura deve ser criada para os dados que serão gerados para o critério potenciais usos.

>qcriasac sac3 st1 poketool pot-usos

Observe na Figura 28 como fica o arquivo das SAC (**ArqSAC**) relativas à ST em questão (**st1**), após a inclusão da **sac3**. Têm-se agora registradas as 3 SAC, com a ferramenta usada e os critérios apoiados em cada um dos casos. O endereço é o do ambiente criado para proteção da sessão de teste (**st1**), como visto na Figura 16.

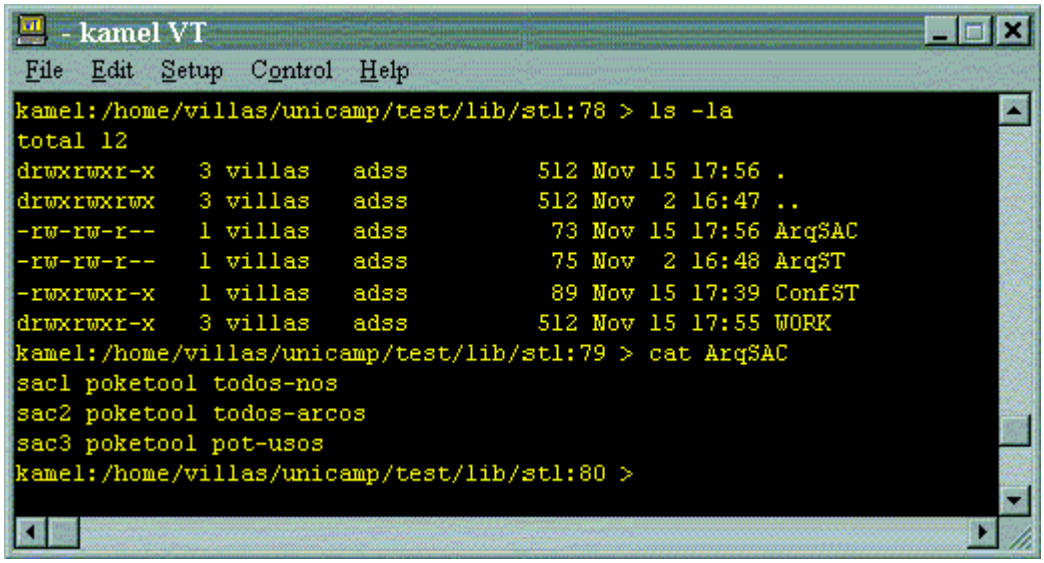

Figura 28 - **ArqSAC** após cadastro das 3 SAC da **st1**

A poketool deve então ser executada para o critério potenciais usos com o caso de teste previsto no plano de teste.

>pokeexec testeprog –fmain (executa o programa instrumentado com os dados de teste escolhidos)

>pokeaval –dmain –pu –ne 1 to 1 (avalia a cobertura obtida para o critério potenciais usos com o dado de teste utilizado)

5.4.12 Criando uma configuração-base para o critério potenciais-usos

Como pode ser observado na Figura 29, a cobertura não foi total, mas de 79,3%. No entanto, o testador concluiu que essa é a melhor cobertura para o critério e, então, decide por aceitar o teste como encerrado. Poderia então, passar para a fase de criação da configuração-base (*baseline*) para o critério potenciais usos, o que é feito com o uso do comando **qcongelasac**, tendo como parâmetros o nome da SAC e o nome da ST em questão.

>qcongelasac sac3 st1

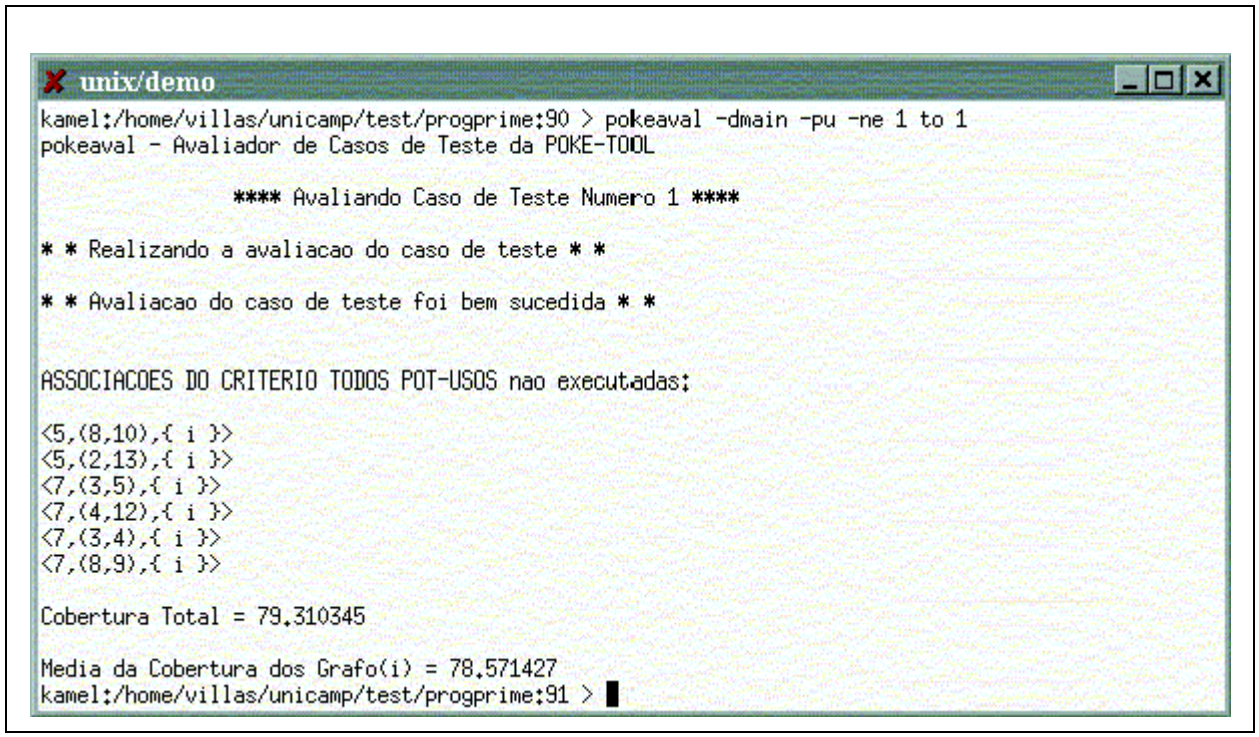

Figura 29 - Execução da avaliação do caso de teste para **sac3**

Neste ponto, segundo o previsto no plano de teste, o teste em si está terminado, com as informações sobre as 3 SAC dessa sessão de teste protegidas para uso posterior. O arquivo **output1.tes** é o arquivo que contém os dados de saída da execução da ferramenta de teste em uso; dessa forma, podemos depreender que ele pertence a todas as configurações-base. Observando a Figura 30, pode-se ver no campo *symbols* desse arquivo no repositório RCS que ele tem a marca das 3 configurações-base criadas (**SACbase-sac1, SACbase-sac2 e SACbase-sac3**) ligadas às respectivas versões salvas do arquivo (**0.1, 0.2 e 0.3**).

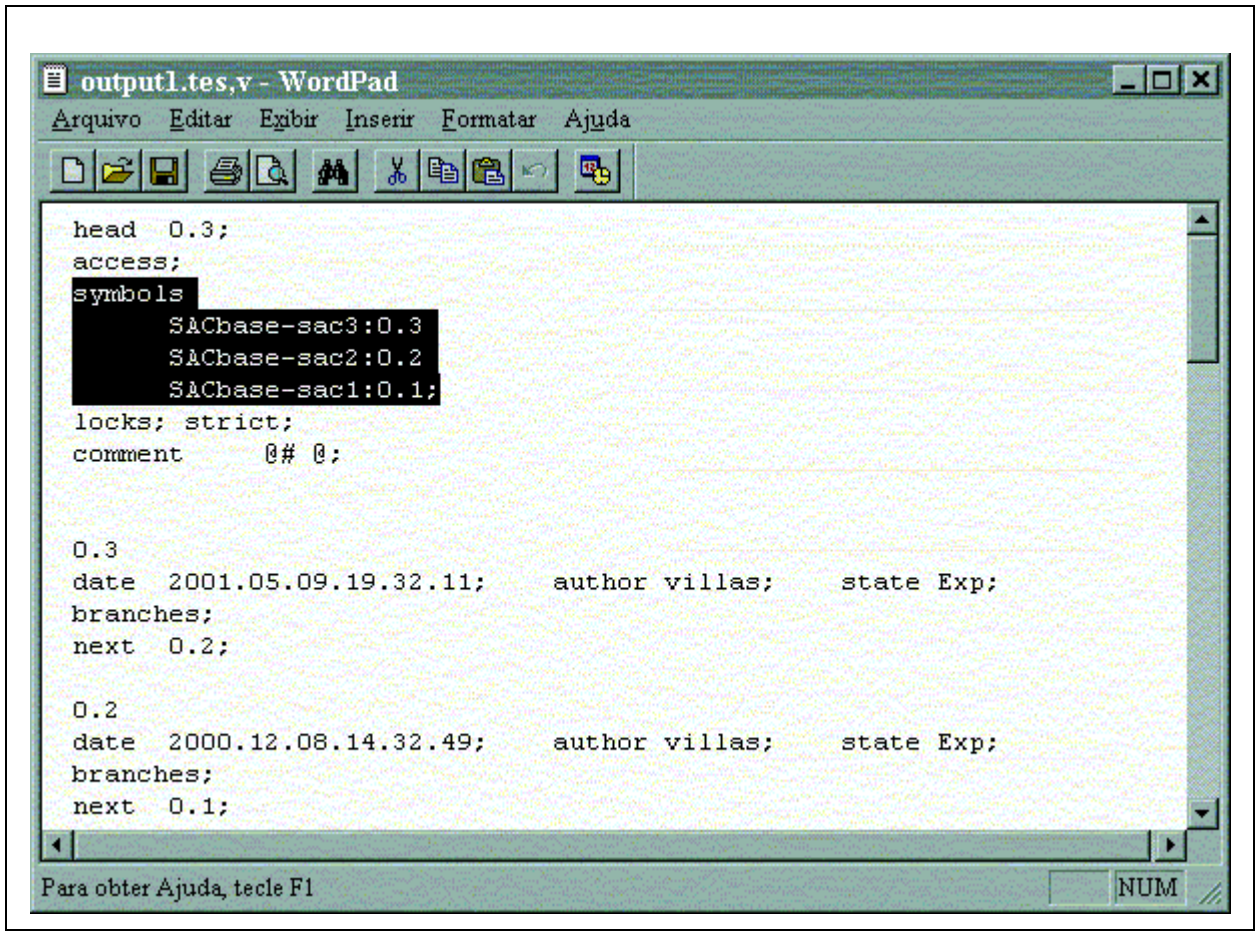

Figura 30 - Visualização do arquivo **output1.tes** no repositório RCS

5.4.13 Recuperando uma SAC

Com essa ST protegida, por meio das configurações-base criadas para cada uma das 3 SAC avaliadas, num outro momento, o testador é capaz de recuperar os dados e executar uma análise para qualquer uma das SAC escolhidas.

Observe na Figura 31 que o diretório da ferramenta de teste está vazio. Caso o testador queira analisar os resultados obtidos com a ferramenta Poke-tool para o critério todos os arcos, ele deve, então, recuperar os dados da segunda SAC que foi salva (**sac2**). Essa operação é feita com o comando **qrecuperasac** tendo como parâmetros o nome da SAC e o nome da ST em questão.

> qrecuperasac sac2 st1

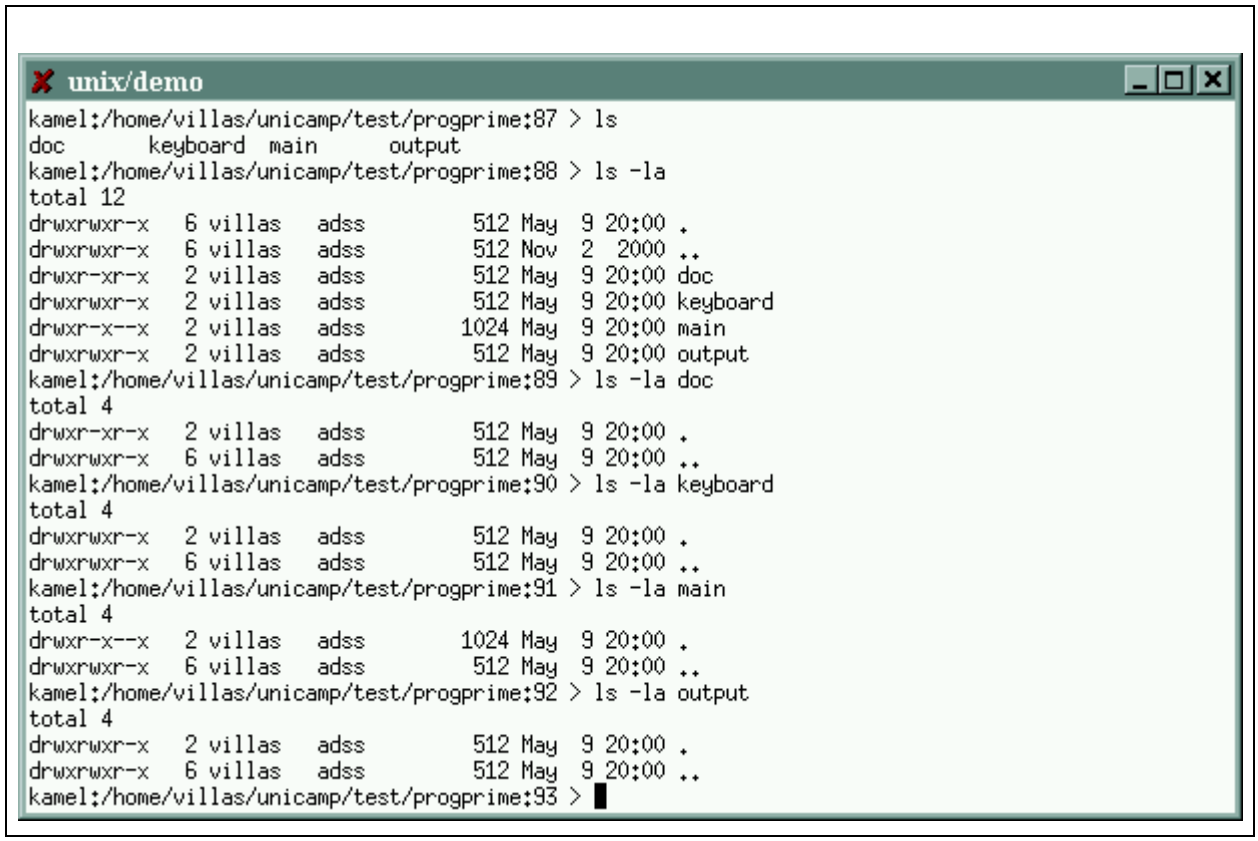

Figura 31 - Diretório da ferramenta de teste com os subdiretórios vazios

Dessa forma, os elementos da **sac2**, marcados com o nome lógico “SACbase-sac2”, são extraídos do repositório RCS associado à **st1** e com as informações de nome e endereço de arquivo de ICT, contidas no arquivo **ArqIC** do ambiente de configuração relacionado à ferramenta e critério em questão, eles são colocados nos locais corretos para que a ferramenta de teste possa ser utilizada para executar a análise requerida. Veja esse resultado na Figura 32 e compare com o visto na Figura 31.

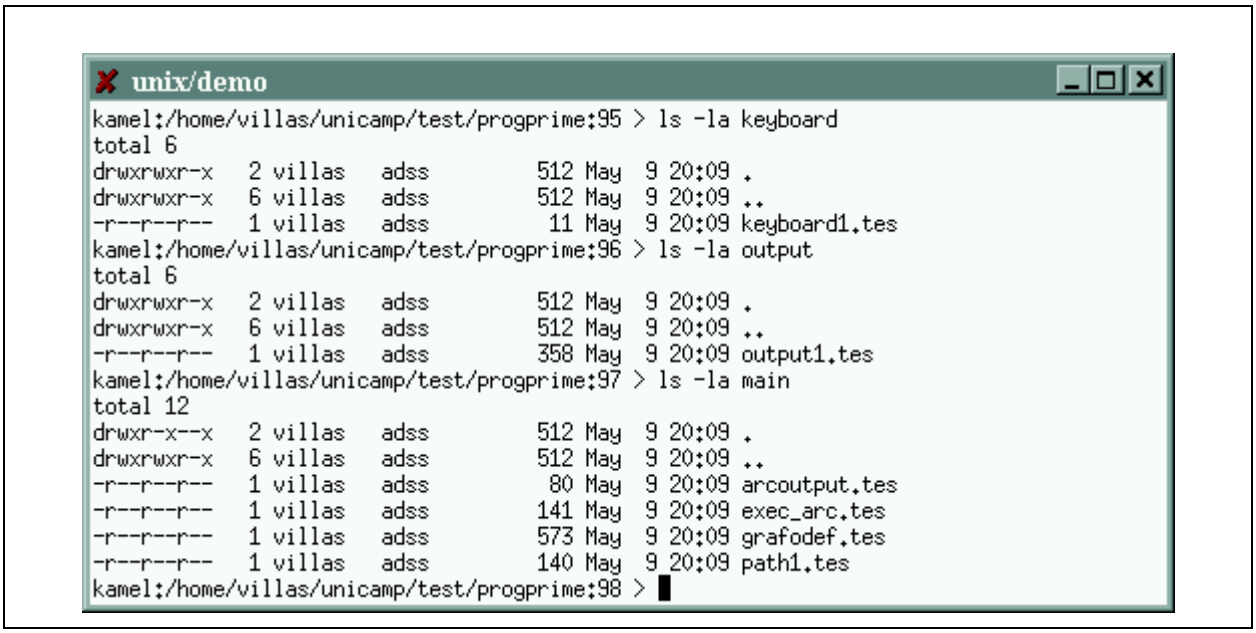

Figura 32 - Diretórios da ferramenta de teste após recuperação dos dados

O exemplo aqui apresentado tem caráter didático, mas a fase de execução e análise de programas é a atividade de maior custo no processo de teste, daí a vantagem em se ter os dados necessários para uma análise futura guardados de forma ordenada.

5.5 Considerações finais

Neste capítulo, foram apresentadas a ferramenta de teste utilizada (Poke-tool), a ferramenta de controle de versões (RCS) e o ambiente de gestão de configuração de sofware para teste desenvolvido para integrar as duas ferramentas e gerenciar os dados gerados na atividade de teste. Foi dado um exemplo prático, com a caracterização de um ICT (utilizando os conceitos apresentados no capítulo anterior) e com o uso do ambiente de controle. No exemplo dado foi mostrada a atuação do testador nesse novo contexto: escolhendo o programa que será testado; escolhendo a ferramenta de teste e os critérios associados; configurando o ambiente operacional (criando os diretórios de trabalho e declarando as variáveis de ambeinte); configurnando o ambiente de gestão de configuração para teste (cadastrando a ferramenta de teste, cadastrando os critérios utlizados, cadastrando o ICT, cadastrando a sessão de teste e cadastrando as sessões de análise de cobertura); executando os testes com a ferramenta de teste escolhida; criando as configurações-base necessárias e por fim recuperando uma configuração-base, mostrando que os dados necessários para uma análise haviam sido recuperados de forma correta. No próximo capítulo serão apresentadas algumas conclusões e propostos alguns possíveis desdobramentos desse trabalho.

Capítulo 6

Conclusões e desdobramentos

6.1 O Problema

O problema aqui levantado está baseado em dois fatos principais: o primeiro é que, em geral, a atividade de teste de unidade é feita de forma pouco sistemática e quase nunca documentada; o segundo é que, mesmo com a ajuda de ferramentas que automatizam parte da atividade de teste, inúmeras vezes a execução dos casos de teste e sua posterior análise são atividades custosas, ou às vezes inviáveis.

6.2 A Proposta de solução

Face a esse problema, propôs-se o uso de conceitos de gestão de configuração de software para apoiar a atividade de teste de unidade, com o objetivo de organizar o ambiente de teste de tal forma que o testador tenha os dados de teste disponíveis para uma possível evolução do software. Num primeiro momento foi caracterizado um item de configuração de teste (ICT), baseado na importância e no custo de se obter esse determinado item. Com isso, dada uma ferramenta de teste e estudando-se seu comportamento, pode-se claramente determinar o conjunto de informações que forma o ICT para essa ferramenta.

Num segundo momento, partiu-se para a automatização do processo de coleta, proteção e recuperação da informação de teste, com o uso de uma ferramenta de gestão de configuração. Um ambiente que integra as ferramentas de gestão de configuração de software para teste, completamente independente da ferramenta de teste usada, foi desenvolvido usando como base os comandos RCS. Para isso, foram desenvolvidos uma série de comandos que possibilitam o testador a:

- configurar o seu ambiente de controle para teste (caracterizando o ICT para determinada ferramenta),
- alterar a configuração do ambiente de teste,
- identificar uma sessão de teste,
- criar uma sessão de análise de cobertura para um dado critério de uma dada ferramenta de teste,
- salvar uma sessão de análise de cobertura e

- recuperar uma sessão de análise de cobertura.

Um exemplo bastante didático foi usado para mostrar a funcionalidade da ferramenta que, com poucos comandos, consegue gerir os dados de teste que são importantes para análise futura e depois recuperá-los.

6.3 Desdobramentos

O ambiente aqui apresentado como solução foi desenvolvido como um protótipo, não seguindo as fases normais de desenvolvimento previstas nas disciplinas de engenharia de software. Portanto, algum trabalho ainda pode ser feito no sentido de melhorá-lo:

1. **Desenvolvimento de uma interface gráfica para a ferramenta:** Embora a falta de uma interface gráfica não se constitua num problema, todos sabem o quanto é mais prático utilizar uma ferramenta que possua uma interface amigável. Com uma interface gráfica, a usabilidade da ferramenta seria melhorada, já que a sintaxe de uso dos comandos estaria embutida na própria interface. Mesmo a questão de seqüência de operações (por exemplo: primeiro se cadastra a ferramenta, depois, uma sessão de teste e só então as sessões de análise de cobertura são cadastradas) poderia ser garantida com o recurso de uso de menus sensíveis ao contexto. Desse modo, o cadastramento de uma SAC só seria possível se a ST correspondente tivesse sido escolhida, por exemplo.

2. **Integração com ferramentas de teste:** No caso das ferramentas de teste Poke-tool[16] e Proteum[20] poderíamos pensar numa integração, não apenas com relação ao repositório, como foi feito no trabalho apresentado, mas também, no que se refere às interfaces. O ambiente de gestão de configuração poderia, desse modo, ser parte do ambiente de teste, criado e gerenciado pela ferramenta de teste. Com essa integração, o testador teria automaticamente ao seu dispor um ambiente de teste controlado, ou seja, como a integração seria feita para cada ferramenta, ela já teria embutida a configuração necessária do ambiente previsto para a gestão de configuração. Ao iniciar uma sessão de teste, ela já seria cadastrada nesse ambiente e ao escolher um dado critério de cobertura, uma sessão de análise de cobertura seria aberta e associada à sessão de teste cadastrada. O testador deveria indicar qual a cobertura esperada e ao atingir essa cobertura a SAC seria automaticamente salva, da mesma forma que tentar parar o teste antes de atingir a cobertura programada seria passível de uma mensagem de aviso, por exemplo. Nesse caso, dever-se-ia pensar numa forma de importar determinados dados de uma

ferramenta de teste para outra, a fim de se realizar comparações.

3. **Estudo para apoio ao teste de regressão:** Mais do que desenvolver novos códigos, a atividade de desenvolvimento de software, atualmente, está mais centrada na manutenção do código hoje existente, afirma Beizer[4]. Ele também afirma que 80% do esforço despendido em produtos de software se concentra na manutenção deles. No processo de manutenção, parte da garantia da qualidade é obtida com o teste de regressão, o qual tem por objetivo testar um software modificado para garantir que essas recentes modificações não tenham alterado, de modo negativo, a funcionalidade desse software[70]. Até 2/3 do custo total de um sistema de software pode ser atribuído ao teste de regressão, afirma Harrold[27], o que com certeza torna essa atividade dispendiosa e praticamente impossível de ser executada a cada pequena modificação executada no software[70].

   Independentemente de haver, ou não, um novo ICT para o teste de regressão, deve-se estudar a necessidade de se estender os conceitos introduzidos pela ferramenta ao apoiar o teste de regressão. Por exemplo, uma sessão de análise de cobertura é encerrada (e, portanto, congelada) ao se atingir a cobertura proposta em seu início, quando o mesmo programa (porém alterado) é novamente testado, a sessão de análise de cobertura será encerrada utilizando-se do mesmo critério de parada, o que acontecerá para um conjunto de dados de teste diferente. O diferencial entre essas configurações-base é justamente o conjunto de dados de teste que foi alterado e, então, pode-se pensar em armazenar apenas a diferença desses conjuntos e criando comandos que possibilitem a recuperação dos conjuntos completos em função da versão do programa em teste que se quer analisar.

4. **Documentação para a ferramenta:** Como explicado o ambiente aqui mostrado é apenas um protótipo usado com a finalidade de validação do modelo de gestão de configuração de software para teste. Para que ele seja transformado em uma ferramenta, faz-se necessário a elaboração de um manual de usuário, com as informações necessárias para sua instalação, configuração e uso. Para uso como protótipo, todas essas informações encontram-se no capítulo 5.

Referências Bibliográficas

[1] ABNT, **Gestão da Qualidade - Diretrizes para a Gestão da Configuração. NBR ISO 10007.** ABNT, 1996.

[2] BABICH, W.A. **Software configuration management - coordination for team productivity.** Addison-Wesley, 1986.

[3] BEIZER, B. **Software testing techniques.** Van Nostrand Reinhold, 1983.

[4] BEIZER, B. **Black-box testing.** John Wiley and Sons, 1995.

[5] BELKHATIR, N., ESTUBLIER, J. **Experience with data base of programs.** In: ACM SIGSOFT/SIGPLAN SYMPOSIUM ON PRACTICAL SOFTWARE DEVELOPMENT ENVIRONMENTS, Palo Alto, CA, dezembro de 1986.

[6] BERSOFF, E.H. et. al. **Software configuration management, an investment in product integrity.** Prentice-Hall, 1980.

[7] BERSOFF, E.H. Elements Of Software Configuration Management. IEEE Transactions on Software Engineering, v. 10, n. 1, p. 79-87, janeiro de 1984.

[8] BLISS, M. Software configuration management - delivering quality software products. Information Systems Management, v. 10, n. 3, p. 35-46, summer 1993.

[9] BONANNI, L.E., SALEMI, C.A. **Source code control systems - user's guide.** Bell Labs., 1981.

[10]BORGES, K. N. et al. **Poke-tool versão clipper — uma ferramenta para suporte ao teste estrutural baseado em análise de fluxo de dados.** In: IX Simpósio Brasileiro de Engenharia de Software, Recife, PE, outubro de 1995.

[11]BRYAN, W. L., SIEGEL, S. G., WHITELEATHER, G. L, Auditing through the software life cycle: a primer. Computer, v. 15, n. 3, p. 57-67, março de 1982.

[12]BUCKLE, J. K. **Software configuration management.** The MacMillan Press Ltd, 1982.

[13]BUDD, T.A. **Mutation analysis: ideas, examples, problems and prospects, chapter of Computer program testing.** North-Holand Publishing Company, 1981.

[14]BUENO, P.M.S. **Geração automática de dados e tratamento de não executabilidade do teste estrutural de software.** Dissertação de Mestrado. FEEC/UNICAMP, 1999.

[15]CAPRETZ, M.A.M. **A software maintenance method based on the software configura-**

tion management discipline. Tese de Doutorado. University of Durham, Inglaterra, 1992.

[16]CHAIM, M.L. **Poke-tool - uma ferramenta para suporte ao teste estrutural de programas baseado em análise de fluxo de dados.** Dissertação de Mestrado. DCA/FEE/UNICAMP, 1991.

[17]CHAIM, M.L. et al. **Poke-tool - versão 2.0 - Manual do usuário.** Relatório técnico, DCA/FEEC/UNICAMP, 1995.

[18]COHEN, E.S. et al. **Version Management in Gypsy.** In: ACM SIGSoft/SIGPlan Software Engineering Symposium on Practical Software Development Environments, Boston, MA, outubro de 1988.

[19]CPqD, UNICAMP, USP/SCar. Workshop do projeto validação e teste de sistemas de operação, Águas de Lindóia, SP, janeiro de 1997.

[20]DELAMARO, M.E. **Proteum - um ambiente de teste baseado na análise de mutantes.** Dissertação de Mestrado. ICMSC-USP/SCar, 1993.

[21]DELAMARO, M.E. **Mutação de interface: um critério de adequação inter-procedimental para teste de integração.** Tese de Doutorado. IFSC-USP/SCar, 1997.

[22]DeMILLO, R.A., LIPTON, R.J. e SAYWARD, F.G. Hints on test data selection: help for the practicing programmer. Computer, v. 11, n. 4, p. 34-41, abril de 1978.

[23]DION, R. Process improvement and the corporate balance sheet. IEEE Software, v. 10, n. 4, p. 28-35, julho de 1993.

[24]FERREIRA, A. B. H. **Novo dicionário Aurélio da língua portuguesa.** Ed. Nova Fronteira, 1986.

[25]FONSECA. R. P. **Suporte ao teste estrutural de programas Fortran no ambiente Poke-tool.** Dissertação de Mestrado. DCA/FEE/UNICAMP, 1993.

[26]GANE, C. **Computer aided software engineering.** Prentice-Hall, 1990.

[27]HARROLD. M. J. **Testing: a roadmap.** In: 22nd International Conference on Software Engineering. Junho, 2000.

[28]HETZEL, W. **The complete guide to software testing**. 2nd edition. QED Info Science Inc., 1988.

[29]HETZEL, W. **Program test methods**. Prentice Hall, 1973.

[30]HORGAN, J.R., MATHUR, A.P. Assessing testing tools in research and education. IEEE Software, v. 9, n. 3, maio de 1992

[31]HUMPHREY, W.S. **Managing the software process.** Addison-Wesley, 1990.

[32]IDE Inc. **Stp/T User´s guide.** Interactive Development Environments, 1994.

[33]IEEE. **IEEE Standard for software test documentation. IEEE Std 829-1998.** IEEE Inc., 1998.

[34]IEEE. **IEEE Standards collection - software engineering.** IEEE Inc., 1993.

[35]IEEE. **IEEE Standard for software configuration management plans. IEEE Std-828-1990**. IEEE Inc., 1990.

[36]IEEE. **IEEE Standard for software review and audits. IEEE Std-1028-1988.** IEEE Inc., 1988.

[37]ISO. **Guidelines for the application of iso 9001 to the development, supply and maintenance of software. IS 9000-3.** ISO*,* 1991.

[38] ISO/IEC. **Quality characteristics and guidelines for their use. IS 9126.** ISO/IEC, 1991.

[39] ISO/IEC. **Software engineering - product evaluation - part 5: process for evaluator's. IS 14598-5.** ISO/IEC, 1996.

[40]ISO/IEC. **Information technology - software life cycle process - configuration management. TR 15846.** ISO/IEC, 1998.

[41]JACKSON, M. **Principles of program design.** Academic Press, 1975.

[42]JONES, C. The economics of software process improvement. Computer, v. 29, n. 1, p. 95-97, janeiro de 1996.

[43]KATZ, R.H., LEHMAN, T.J. Database support for versions and alternatives of large design files. IEEE Transactions on Software Engineering, v. 10, n. 2, p. 191-200, março de 1984.

[44]LEBLANG, D.B., CHASE, R.P. **Computer-aided software engineering in a distributed workstation envirnment.** In: ACM SIGSOFT/SIGPLAN SYMPOSIUM ON PRACTICAL SOFTWARE DEVELOPMENT ENVIRONMENTS, Pittsburgh, PA, abril de 1984.

[45]LORI, L.A. Physical integrity in a large segmented database. ACM Transaction on Database Systems, v. 2, n. 1, p. 91-104, março de 1977.

[46]LEITÃO. P. S. J. **Suporte ao teste estrutural de programas COBOL no ambiente Poke-tool.** Dissertação de Mestrado. DCA/FEE/UNICAMP, 1992.

[47]MALDONADO, J.C. **Critérios potenciais usos: uma contribuição ao teste estrutural de software.** Tese de Doutorado. DCA/FEE/UNICAMP, 1991.

[48]MALDONADO, J.C. et al. **Introdução ao teste de software.** In: XIV SIMPÓSIO

BRASILEIRO DE ENGENHARIA DE SOFTWARE, João Pessoa, PB, outubro de 2000.

[49]McCABE, T. A Complexity measure. IEEE Transactions on Software Engineering, v. 2, n. 4, p. 308-320, dezembro de 1976.

[50]Mercury Interactive. **XRunner user´s guide.** Mercury Interactive Inc., 1994.

[51]MYERS, G. **The art of software testing.** John Wiley and Sons, 1979.

[52]MUSA, *Metodologia Unificada de Sistemas Aplicativos*, Versão 1.0, julho 1992, Grupo Específico de Metodologias de Desenvolvimento de Sistemas da TELEBRÁS, Departamento de Coordenação de Informática da TELEBRÁS.

[53]NAKAGAWA. E. Y. et al. **Aspectos de projeto e implementação de interfaces gráficas do usuário para ferramentas de teste.** In: Workshop do Projeto de Validação e Teste de Sistemas de Operação, Águas de Lindóia, SP, janeiro de 1997.

[54]PAULK, M.C. et al. **Capability Maturity Model for Software - ver 1.1. CMU/SEI-93-TR-24.** CMU/SEI, 1993.

[55]PERRY, W. **Effective methods for software testing.** John Wiley and Sons, 1995.

[56]PFLEEGER, S. L., HATTON, L. Investigating the influence of formal methods. Computer, v. 30, n. 2, p. 33-43, fevereiro de 1997.

[57]PRESSMAN, R. S. **Software engineering - a practitioner's approach**. 5th. ed. McGraw-Hill, 2001.

[58]SCHACH, S.R. **Software engineering.** Asken Associates Inc.,1990.

[59]SEPIN. **Qualidade e produtividade no setor de software brasileiro 2001.** MCT/SEPIN, 2002.

[60]SOFTOOL. **CCC/manager - data base administrator's guide - ver 2.0.** Softool Corp., Goleta, CA, 1991.

[61]SOFTWARE RESEARCH Inc. **STW/coverage tool suite for C. User manual, Release 2**, maio de 1994.

[62]SOUZA, S.R.S. **Avaliação do custo e eficácia do critério análise de mutantes na atividade de teste de programas.** Dissertação de mestrado. ICMC/USP-SCar, 1996.

[63]TICHY, W.F. RCS -- a system for version control. Software-Practice & Experience, v. 15, n. 7, p. 637-654, julho de 1985.

[64]TOMAYKO, J.E. **Software configuration management. SEI-CM-4-1.4.** SEI, 1990.

[65]VICTORELLI, E.Z. **Controle de versões e configurações em ambientes de desenvolvi-**

mento de software. Dissertação de Mestrado. DCC/IMECC/UNICAMP, 1990.

[66]VICTORELLI, E.Z., MAGALHÃES, G.C. **MVC: um modelo para controle de versões e configurações.** In: V Simpósio Brasileiro de Engenharia de Software, Ouro Preto, MG, outubro de 1991.

[67]VILELA. P. R. S. **Critérios potenciais usos de integração: definição e análise.** Tese de Doutorado. DCA/FEEC/UNICAMP, 1998.

[68]VIOLATO, C.A. Sistemas de suporte à operação. Revista TELEBRÁS, v. 16, n. 56, outubro de 1992.

[69]WOHLWEND, H., ROSENBAUM, S. Schlumberger's software improvement program. IEEE Transactions on Software Engineering, v. 20, n. 11, p. 833-839, novembro de 1994.

[70]WONG, E., MALDONADO. J. C., DELAMARO, M. E. **Reducing the cost of regression testing by using selective mutation.** In: VIII Conferência Internacional de Tecnologia de Software, Curitiba, PR, junho de 1997.

[71]XAVIER, P. SAGRE/SGE/SADAN - Sistemas de suporte à operação. Revista TELEBRÁS, v. 16, n. 56, outubro de 1992.

[72]YOURDON, E. **Modern structured analysis.** Englewood Cliffs, 1989.

[73]YOURDON, E., CONSTANTINE, L. L. **Structured design: fundamentals of a discipline of computer program and system design.** Englewood Cliffs, 1975.

Anexo A - Metadocumento de PLANO DE TESTE

1 Introdução

1.1 Objetivos

Descrever os objetivos deste Plano de Teste, tais como:

- Detalhar as atividades requeridas para preparar e conduzir os testes;
- Comunicar a todas as partes envolvidas as tarefas sob sua responsabilidade e seus cronogramas;
- Definir as fontes de informação utilizadas para preparar o plano; e
- Definir as ferramentas de teste e o ambiente necessário para a condução dos testes.

1.2 Escopo

Identificar o nível de teste coberto pelo Plano (UI, integração, sistema ou aceitação) bem como a funcionalidade e os procedimentos por ele abrangidos.

1.3 Referências

Listar os documentos (nome e código) utilizados como fonte de informação para a elaboração do Plano. Tipicamente, deverão ser incluídos:

- Plano de Desenvolvimento (PS);
- Plano de Garantia de Qualidade (PQ);
- Plano de Gerência de Configurações e Versões (PV);
- Padrões Adotados; e
- Políticas Relevantes.

2 Itens de Teste

Para o Plano de Teste de Sistema esta seção só será completada após o término do Projeto Físico.

Nesta seção serão identificados os itens de software que serão testados. Para cada item deverão ser identificados:

- Tipo (Arquivo Fonte, Arquivo Executável, Utilitários etc.);
- Identificação (Código e Nome);
- Localização (Biblioteca, Módulo de Implementação etc.);
- Versão; e
- Meios e Procedimentos para obtenção dos itens de software (descompactação, transferência de fita ou disco etc.)

2.1 Módulos de Teste

Identifique os componentes do sistema de aplicação a serem testados.

2.2 Recursos Externos

Identifique os componentes externos a serem utilizados nos testes (Gerenciador de Banco de Dados, Bibliotecas etc.).

2.3 Documentos Referidos

Identifique dentre os documentos de projeto abaixo, aqueles que se aplicam ao nível de teste coberto pelo Plano:

- Especificação de Objetivos Requisitos (OR)
- Especificação de Características de UI (CA)
- Especificação de Definição das Interfaces (EC)
- Descrição Funcional (DF)

- Projeto de Interface de Usuário (PU)
- Manual de Usuário (MU)
- Manual de Operação em Produção (MO)
- Qualquer relatório de problema ocorrido durante aplicações anteriores dos testes

3 Características a serem Testadas

Identificar todas as características, ou combinações de características, do software a ser testado. Para cada grupo indicar:

- Descrição da característica
- Referência ao documento de Projeto de Teste associado

4 Características não Testadas

Identificar todas as características, ou combinações de características de software, que **não** serão testadas, explicando o motivo da sua exclusão.

5 Métodos de Teste

Para cada característica ou combinação de características especificada na seção 3 inclua uma subseção contendo as seguintes informações:

- Atividades de teste a serem executadas;
- Técnicas a serem utilizadas;
- Ferramentas a serem empregadas;
- Grau de abrangência desejado;
- Técnicas utilizadas para julgar o grau de abrangência;
- Critérios de término de teste de modo a garantir a abrangência desejada;

- Restrições às atividades de teste, tais como, disponibilidade de recursos, disponibilidade de itens de teste e datas limites; e

- Técnicas a serem utilizadas para rastreamento dos requisitos a serem testados.

Os itens acima devem ser descritos com um grau de detalhamento suficiente para permitir a identificação das tarefas de teste e estimar o tempo necessário para sua execução.

6 Critérios de Aceitação

Especifique o(s) critério(s) a ser(em) utilizado(s) para determinar se cada item de teste obteve, ou não, sucesso nos testes a que foi submetido.

7 Critérios de Suspensão e Retomada de Testes

Especificar o(s) critério(s) utilizados para suspender parcialmente, ou totalmente, a atividade de teste associada a este plano. Especificar também a(s) atividade(s) que deve(m) ser repetidas quando da retomada do teste.

8 Produtos a serem Gerados

Identifique todos os produtos a serem gerados pelas atividades de teste, incluindo:

- Documentos. Os seguintes documentos podem ser gerados:

  Plano de Teste (PN);

  Projetos de Teste (PJ);

  Casos de Teste (CS):

  Procedimentos de Teste (PR);

  Registros de Teste (RG);

  Relatórios de Problemas de Teste (RT); ou

  Relatórios de Resumo de Teste (RE).

- Rotinas de teste (drivers e stubs)

9 Tarefas de Teste

Identificar todas as tarefas necessárias para preparar e executar os testes. Identificar também as dependências inter-tarefas e quaisquer habilidades especiais que se façam necessárias. Veja abaixo um exemplo de parte de uma lista de tarefas de um Plano de Teste de Sistema para um sistema de folha de pagamento.

Tarefa	**Tarefas Precedentes**	**Habilidades Especiais**	**Responsabilidade**	**Esforço (h/h)**	**Data Limite**
1. Preparar Plano de Teste	Especificação de requisitos		Gerente de Teste e Analista de teste senior	4	21/01/93
2. Preparar Projeto de Teste	Tarefa 1	Conhecimento dos procedimentos de pagamento da corporação (conhecimento do Negócio)	Analista de teste senior	9	01/04/93
3. Preparar Casos de Teste	Tarefa 2		Analista de teste	4	15/04/93

10 Ambiente de teste

Especificar as propriedades necessárias e desejadas para o ambiente onde será executado o teste. Nesta especificação devem estar descritos:

- Características físicas do equipamento a ser utilizado

- Software de sistema e de comunicação
- Modo de operação (p.ex. *stand-alone*)
- Ferramentas de teste
- Outras necessidades (publicações, espaço físico etc.)

11 Responsabilidades

Identificar os grupos responsáveis por gerenciamento, projeto, preparação, execução, observação, verificação, e solução dos problemas. Em adição deve identificar os grupos responsáveis por fornecer os itens de teste e o ambiente de teste.

Estes grupos podem incluir os projetistas, os testadores, os operadores, os representantes do cliente, o pessoal de suporte, o grupo de administração de dados e o grupo de qualidade.

12 Formação da Equipe de Teste

Especificar a composição da Equipe de Teste em termos de número de membros e do grau de proficiência de cada um. Em caso de necessidade, definir o tipo de treinamento a que eles devam ser submetidos.

13 Cronograma

Incluir os marcos de teste descritos no cronograma geral do projeto, bem como todos os eventos de entrega de itens para teste. Estimar o tempo necessário para cada tarefa e cada marco de teste, e o período de uso de cada recurso utilizado. Esta seção possui estreita relação com o que é descrito na seção 9.

14 Riscos e Plano de Contingência

Identificar as hipóteses de risco do plano e especificar planos de contingência para cada uma, identificando também seu responsável (p.ex., o atraso na entrega de itens de teste tem influência sobre o cronograma, ou o atraso na compra de uma ferramenta, ou ainda a não execução de um treinamento específico).

Anexo B - Metadocumento de PROJETO DE TESTE

1 Características a serem Testadas

Identificar, nesta seção, as características e combinações de características do módulo que é objeto deste documento. Devem ser incluídas referências cruzadas aos documentos de projeto (OR, DF, PU, CA, EC) que contêm os requisitos das características a serem testadas.

1.1 Características Individuais

Exemplos:

Processamento de Dígitos

Processamento de Sinal

Processamento de Ponto Decimal

Processamento de Vírgulas

1.2 Combinações de Características

Exemplos:

Ponto Decimal e Sinal

Sinal e Vírgulas

Ponto Decimal e Vírgulas

Sinal, Ponto Decimal e Vírgulas

2 Detalhamento do Método

Detalhar o método descrito no Plano de Teste. Incluir as técnicas específicas de teste que serão usadas. Descrever o método de análise dos resultados (p.ex. inspeção visual ou programa comparador).

Especificar os resultados de qualquer análise que forneça uma “base lógica’’ para escolha de casos de teste.

Resumir atributos comuns a quaisquer casos de teste, identificando restrições de entradas, necessidades ambientais compartilhadas, quaisquer requisitos procedimentais especiais compartilhados, e quaisquer casos de dependências compartilhadas.

Exemplo:

As características individuais do módulo serão testadas primeiro com entradas válidas e inválidas. Todas as combinações serão então testadas.

Deverá ser escrito um programa para exercitar os testes. Um arquivo de exercício deverá ser criado contendo dois campos por registro: o primeiro conterá um valor único de entrada para teste, e o segundo armazenará o resultado do teste. Esse programa de teste lerá cada registro, passará o valor de entrada correspondente ao módulo e rescreverá o registro colocando o valor resultante no segundo campo.

Antes da execução dos testes, deverá ser gerado um arquivo de casos de teste com o mesmo formato do arquivo de exercício. Seus registros conterão os valores de entrada juntamente com os resultados antecipados. Após a execução do teste, o arquivo de exercício deverá ser comparado com o arquivo de casos de teste. Um utilitário de comparação de arquivos deverá detectar quaisquer diferenças.

Desde que a geração de todos os valores de entrada é impraticável, a abrangência do conjunto de testes deverá ser avaliada através dos seguintes critérios:

- *1) Cobertura dos requisitos — Cada um dos requisitos foi coberto?*
- *2) Cobertura de projeto — Cada uma das especificações funcionais foi coberta?*
- *3) Cobertura de domínio — Cada uma das restrições de entrada foi testada? Valores representativos foram incluídos? Todas as mensagens de erro foram geradas?*
- *4) Cobertura de ramificação — Todo caminho de execução foi exercitado?*
- *5) Cobertura de comando — Todo comando do módulo foi executado pelo menos uma vez?*

Deverão ser criadas Listas de Verificação (“Check Lists’’) para avaliar os critérios 1, 2 e 3. As ferramentas de teste existentes deverão ser utilizadas para avaliar os critérios 4 e 5.

O conjunto de testes deve satisfazer cada um dos critérios acima especificados, pelo menos uma vez.

Os seguintes critérios deverão ser adotados para seleção de casos de testes:

- *Não mais que 14 dígitos inteiros*
- *Não mais que 4 dígitos fracionários*
- *Não mais que 1 ponto decimal*
- *Entre 1 e 3 dígitos contíguos à esquerda de cada vírgula*
- *Exatamente 3 dígitos contíguos à direita de cada vírgula*
- *Nenhuma vírgula depois do ponto decimal*

3 Identificação dos Testes

Listar o identificador e uma breve descrição para cada caso de teste. Um caso de teste poderá ser identificado em mais de um Projeto de Teste.

Listar o identificador e uma breve descrição de cada Procedimento de Teste associado com este Projeto de Teste.

3.1 Casos de Teste

Exemplos:

1. *Dígitos somente*
    - *Válidos*
        - *•14 dígitos inteiros (descrito no documento NNE.CS.001)*

•6 dígitos inteiros centrado (descrito no documento NNE.CS.002)

• 1 dígito inteiro justificado à esquerda (descrito no documento NNE.CS.003)

- *Inválidos*

  • 15 dígitos inteiros (descrito no documento NNE.CS.010)

2. *Sinal*

   - *Válidos*

     • 14 dígitos inteiros justificados à direita (descrito no documento NNE.CS.020)

   - *Inválidos*

     • Sinal embutido (descrito no documento NNE.CS.030)

3.2 Procedimentos de Teste

Listar o identificador e dar uma breve descrição dos Procedimentos de Teste associados a este Projeto de Teste.

4 Critérios de Aceitação

Para cada característica ou combinação de características testada, especificar os critérios utilizados para determinar se a característica foi aprovada ou não nos testes.

Exemplo:

Cada característica deve passar, com sucesso, por todos os seus casos de teste para ser aprovada.

Anexo C - Metadocumento de CASOS DE TESTE

1 Itens de Teste

Identificar e descrever resumidamente os itens de software e as características que serão exercitadas por este Caso de Teste.

Para cada item devem ser feitas referências aos seguintes documentos, quando as características por ele exercitadas estiverem neles descritas:

- Especificação de Objetivos Requisitos (OR);
- Especificação de Características de UI (CA);
- Especificação de Definição das Interfaces (EC);
- Descrição Funcional (DF);
- Projeto de Interface de Usuário (PU);
- Manual de Usuário (MU);
- Manual de Operação em Produção (MO).

Exemplo:

Subrotina de Normalização de Expressão Numérica (SR-0304.1.1). Esta rotina extrai sinal, vírgulas e pontos decimais de expressões numéricas. Os requisitos funcionais desta rotina estão descritos na seção 4.2 do Documento de Descrição Funcional (DF-0104.1.1).

2 Especificação das Entradas

Especificar as entradas necessárias para execução do caso de teste. Elas podem ser especificadas por valor (incluindo a tolerância quando necessário) ou por nomes (no caso de tabelas ou arquivos). Identificar a base de dados, arquivos, mensagens de terminal, áreas de memória e valores passados pelo sistema operacional.

Especificar os requisitos de relacionamento entre entradas (p.ex. temporização).

Exemplo:

Arquivo CS001.DAT

3 Especificação das Saídas

Especificar todas as saídas e características requeridas (p.ex. tempo de resposta) fornecendo os valores exatos (com tolerância, quando houver) para cada saída.

Exemplo:

VALOR-NUMERICO-ALINHADO = +12340000

CODIGO-RETORNO = "NORMALIZACAO-OK''

NUMERO-DIGITOS-INTEIROS = 4

4 Ambiente Necessário

Hard-ware:	Especificar as características e configurações do hardware necessário para execução do caso de teste (p.ex. monitor VGA).
Soft-ware:	Especificar o software de sistema e de aplicação necessários para a execução do caso de teste, tais como sistema operacional, compiladores, simuladores e outras ferramentas.
Outros:	Qualquer outra necessidade particular (p.ex. pessoal com treinamento especializado).

Exemplo:

O programa de teste PE001.EXE é necessário para executar este caso de teste.

5 Procedimentos Especiais

Descrever qualquer restrição especial sobre os procedimentos de teste que executam este caso de teste. Estas restrições podem ser dos seguintes tipos:

- Preparação especial;
- Intervenção do operador;
- Procedimentos de determinação de saídas; ou
- Epílogos (*wrap-up*) especiais.

6 Dependência de Testes

Listar os identificadores dos outros casos de teste que devem ser executados antes deste caso de teste. Resuma a natureza das dependências.

Anexo D - Metadocumento de PROCEDIMENTOS DE TESTE

1 Objetivos

Descrever os objetivos deste procedimento, isto é, os passos necessários à execução do conjunto de casos de testes cobertos por este procedimento. Identificar (fazendo referência) os casos de teste que serão executados. Também devem ser identificadas as seções relevantes dos documentos de projeto que descrevem os requisitos, a estrutura e a funcionalidade dos itens de teste (OR, DF, PU, CA, EC).

2 Requisitos Especiais

Identificar quaisquer requisitos especiais que são necessários à execução deste procedimento. Podem estar incluídos requisitos de ambiente, de habilidades da equipe de teste e ações prévias necessárias à execução deste procedimento.

3 Passos do Procedimento

Incluir as seguintes subseções quando aplicáveis.

3.1 Registro

Identificar o documento de Registro de Teste associado a este procedimento.

3.2 Preparação

Descrever a seqüência de ações necessárias para preparar a execução do procedimento.

3.3 Iniciação

Descrever as ações necessárias para dar início à execução do procedimento.

3.4 Procedimentos

Descrever as ações necessárias durante a execução do procedimento.

3.5 Método de Medição

Descrever como as medições do teste serão realizadas (p.ex. como o tempo de resposta será medido).

3.6 Interrupção

Descrever as ações necessárias para a suspensão do teste, quando ações não planejadas assim o determinarem.

3.7 Reiniciação

Identificar quaisquer pontos de reinicio e descrever as ações necessárias ao reinicio do procedimento em cada um destes pontos.

3.8 Finalização

Descrever as ações necessárias para o encerramento da execução normal do procedimento.

3.9 Epílogo

Descrever as ações necessárias para restaurar o ambiente após a execução deste procedimento.

3.10 Contingências

Descrever as ações necessárias para tratar eventos anômalos que ocorram durante a execução do procedimento.

Anexo E - Metadocumento de REGISTROS DE TESTE

1 Descrição

As informações que se aplicarem a todos os eventos relatados na próxima seção, devem ser incluídas nesta seção. No mínimo as seguintes informações devem ser incluídas:

- Itens de software sendo testados (com versão e nível de revisão).
- Atributos do ambiente onde o teste está sendo executado. Incluir a identificação da instalação, o hardware utilizado (p.ex. quantidade de memória, modelo da CPU, dispositivos de armazenamento de informação etc.), o software utilizado e os recursos disponíveis.

2 Registro de Atividades e Eventos

Para cada evento, incluindo o início e o fim das atividades de teste, registrar a data e a hora da ocorrência, junto com a identidade do autor.

As seguintes informações devem ser incluídas, quando aplicáveis, para cada evento:

1. Descrição da execução

Registrar o identificador do Procedimentos de Teste em execução. Identificar todo o pessoal presente na execução indicando a função de cada um (testador, observador, operador etc.).

2. Resultados obtidos

Para cada execução registrar os resultados visualmente observáveis (p.ex. mensagens de erro, interrupções, etc.). Registrar a localização de cada saída (arquivo, fita etc.) e se houve sucesso, ou não, na execução do teste.

3. Informações de ambiente

Registrar qualquer condição ambiental específica para o evento (p.ex. substituição de hardware).

4. Eventos anômalos

Registrar o que aconteceu antes e depois da ocorrência de um evento inesperado. Registrar as circunstâncias que envolvem a inabilidade de se iniciar a execução de um procedimento de teste ou a incapacidade de completá-lo (p.ex. falta de energia ou problemas no sistema operacional).

5. Identificadores dos relatórios de problemas

Registre o identificador do Relatório de Problemas de Teste associado ao evento, sempre que ele tiver sido gerado.

Anexo F - Metadocumento de RELATÓRIO DE PROBLEMAS DE TESTE

1 Resumo

Descrever de modo resumido o incidente ocorrido. Identificar os itens de teste envolvidos (com versão e nível de revisão). Devem ser feitas referências aos procedimentos de teste, casos de teste e registros de teste associados.

2 Descrição do Problema

Fornecer uma descrição do problema, a qual deve incluir os seguintes itens:

- Entradas;
- Resultados esperados;
- Anomalias;
- Data e hora;
- Passo do procedimento;
- Ambiente;
- Tentativas de repetição;
- Testadores; e
- Observadores.

Atividades relacionadas e observações que possam ajudar a isolar e corrigir a causa do problema devem ser incluídas. Registrar, por exemplo, variações com relação ao procedimento de teste utilizado ou execução de outros casos de teste.

3 Impacto

Caso seja conhecido, indicar que impacto este problema pode vir a causar no plano de teste, no projeto de teste, nos procedimentos de teste, ou nos casos de teste.

Anexo G - Metadocumento de RELATÓRIO DE RESUMO DE TESTE

1 Resumo

Resumir a avaliação dos itens de teste. Identificar os itens de software testados (com versão e nível de revisão). Indicar o ambiente no qual as atividades de teste foram executadas.

Para cada item de teste, fornecer referências (quando existirem) aos seguintes documentos:

- Plano de Teste (PN);
- Projeto(s) de Teste (PJ);
- Procedimentos(s) de Teste (PR);
- Registros(s) de Teste (RG); e
- Relatório(s) de Problemas de Teste (RT).

2 Desvios

Relatar qualquer desvio dos itens de teste com relação à suas especificações de projeto. Indicar qualquer variação com relação ao Plano de Teste, Projetos de Teste ou Procedimentos de Teste, explicando a razão desta variação.

3 Análise de Abrangência

Avaliar a abrangência do processo de teste com relação aos critérios estabelecidos no Plano de Teste (caso exista um). Identificar características, ou combinações de características, as quais não foram suficientemente testadas e explicar as razões.

4 Resumo dos Resultados

Resumir os resultados do teste. Identificar todos os incidentes resolvidos e relatar resumidamente sua solução. Identificar todos os incidentes não resolvidos.

5 Avaliação dos Testes

Fornecer uma avaliação geral de todos os itens de teste incluindo suas limitações. Esta avaliação deve ser baseada nos resultados dos testes e no critério adotado para julgar se um item obteve, ou não, sucesso no teste. Uma estimativa de risco de falha pode ser incluída.

6 Resumo das Atividades

Resumir as principais atividades e eventos de teste. Descrever resumidamente dados sobre o uso de recursos, como tempo total de uso de máquina, total de pessoal, tempo de sala, etc.

Anexo H - *Scripts* do ambiente de GCST

1 Configuração do ambiente de controle (qconfigura)

```
#!/bin/sh
##############################################################################
# Nome: qconfigura
#
# Entradas: Dados para configuracao da sessao de teste
#
# Saidas: Estrutura de diretorios e arquivos de configuracao do ambiente
#         de teste
#
# Funcao: Criar diretorios e arquivos com os dados de configuracao do ambiente #         de teste
#
##############################################################################

Usage(){
        PROG=`basename $0`
        echo "Programa iterativo para configuracao da sessao de teste"
        }

## Inicio

## Se nao existe o diretorio CONF entao cria
##
#CONFDIR=$HOME/unicamp/test/conf;
#if test ! -d $CONFDIR
#         then mkdir $CONFDIR;
```

```
#
#fi

echo "Novo cadastro [N] ou altercao de cadastro [A]?";
read resp;
if test "$resp" = N

### E' um novo cadastro
###
then

crtfer=S;
while test "$crtfer" = S
do
        echo "Insira nome da ferramenta usada-";
        read nomfer;
        FERDIR=$CONFDIR/$nomfer;
        if test ! -d $FERDIR
                then mkdir $FERDIR;
        fi
        critbol=S;
        while test "$critbol" = S
        do
                echo "Deseja cadastrar um criterio? [S] ou [N].";
                read resp;
                if test "$resp" = S
                        then  echo "Insira o nome do criterio-";
                                read crit;
                                if test -d $crit
```

```
                    then    echo "Criterio ja cadastrado!!!";
                    else    echo "$crit" >> $FERDIR/Ferconf;
                            CRITDIR=$FERDIR/$crit;
                            mkdir $CRITDIR;
                            ctric=S;
                            while test "$ctric" = S
                            do
                            if test "$ctric" = S
                            then    echo "Insira o IC de teste-";
                                    read ic;
                                    echo "$ic" >> $CRITDIR/ArqIC;
                                    echo "Insira o endereco desse IC de teste -";
                                    read ic;
                                    echo "$ic" >> $CRITDIR/ArqIC;
                                    echo "Insira a descricao do IC-";
                                    read ic;
                                    echo "$ic" >> $CRITDIR/ArqIC;
                            fi
                            echo "Deseja incluir mais IC? [S] ou [N].";
                            read ctric;
                            done
                    fi
            else  critbol=N;
        fi
    done
echo "Deseja cadastrar outra ferramenta? [S] ou [N].";
read crtfer;
done
```

```
### E' uma alteracao de cadastro
###

else
echo "Favor indicar a ferramenta de teste usada:";
read ferr;
if test ! -d $CONFDIR/$ferr
        then echo "Ferramenta de teste nao cadastrada!";
            exit 1;
fi
echo "Favor indicar o criterio ou tecle [T] para exclusao total:";
read crit;
if test "$crit" != T
        then    if test ! -d $CONFDIR/$ferr/$crit
                        then    echo "Criterio nao cadastrado!";
                                exit 1;
                        else    rm -fr $CONFDIR/$ferr/$crit;
                                echo "O criterio "$crit" foi retirado do cadastro da ferramenta
"$ferr"";
                fi
        else    rm -fr $CONFDIR/$ferr;
                echo "A ferramenta "$ferr" foi retirada do cadastro.";
fi
fi
```

2 Script de criação da sessão de teste (qcriast)

```
#!/bin/sh
################################################################
# Nome: qcriast
```

```
#
# Entradas: Dados de cadastro da sessao de teste (ST)
#
# Saidas: Um arquivo de cadastro da ST
#
# Funcao: Criar um arquivo ArqST e nele inserir o
#         responsavel, a ferramenta, o criterio e quando
#         houver plano e registro de teste e programas
#         auxiliares
#################################################################

Usage(){
        PROG=`basename $0`
        echo "Cria uma Sessao de Teste (ST)"
}

### Programa principal ######

### Le o nome da ST

ctrst=0;
while test $ctrst = 0
do
        echo "Indique o nome da sessao de teste (ST):";
        read ST;
        STDIR=$LIBDIR/$ST;
        if test -d $STDIR
                then    echo "A ST $1 ja' existe!";
                else    mkdir $STDIR;
```

```
                ctrst=1;
        fi
done

### Le o nome do responsavel
resp=$USER;
echo "+$resp" >>$STDIR/ArqST;

### Le o nome da ferramenta

crtfer=S;
while test "$crtfer" = S
do
        echo "Indique o nome da ferramenta:";
        read ferram;
        if test -d $CONFDIR/$ferram
                then    echo "$ferram" >> $STDIR/ArqST;
                else    echo "Ferramenta nao cadastrada, favor cadastrar!";
                        rm $STDIR/ArqST;
                        rmdir $STDIR;
                        exit 1;
        fi

### Le o nome do criterio

        crtcri=S;
        while test "$crtcri" = S
        do
                echo "Indique o nome do criterio:";
```

```
                read crit;
                if test -d $CONFDIR/$ferram/$crit
                        then    echo "$crit $ferram" >> $STDIR/ArqST;
                        else    echo "Criterio nao cadastrado, favor cadastrar!";
                                rm $STDIR/ArqST;
                                rmdir $STDIR;
                                exit 1;
                fi
                echo "Deseja cadastrar outro criterio? [S] ou [N].";
                read crtcri;
                done

echo "Deseja cadastrar outra ferramenta? [S] ou [N].";
read crtfer;
done

### Cadastra plano de teste

echo "Possui Plano de Teste? [S] ou [N].";
read resp;
if test "$resp" = S
        then  echo "Nome do arquivo de Plano de Teste-";
                read pltes;
                echo $pltes >> $STDIR/ConfST;
                echo "Endereco do diretorio do Plano de Teste-";
                read endpltes;
                echo $endpltes >> $STDIR/ConfST;
                descpltes="Documento do Plano de Teste";
                echo $descpltes >> $STDIR/ConfST;
```

```
fi

### Cadastra casos de teste

echo "Possui documento de casos de teste? [S] ou [N].";
read resp;
if test "$resp" = S
        then  echo "Nome do arquivo de Casos de Teste-";
                read castes;
                echo $castes >> $STDIR/ConfST;
                echo "Endereco do diretorio do Casos de Teste-";
                read endcastes;
                echo $endcastes >> $STDIR/ConfST;
                descastes="Documento de casos de teste";
                echo $descastes >> $STDIR/ConfST;
fi

### Cadastra relatorio de teste

echo "Possui documento relatorio de teste? [S] ou [N].";
read resp;
if test "$resp" = S
        then  echo "Nome do arquivo do Relatorio de Teste-";
                read reltes;
                echo $reltes >> $STDIR/ConfST;
                echo "Endereco do diretorio do Relatorio de Teste-";
                read endreltes;
                echo $endreltes >> $STDIR/ConfST;
                descreltes="Documento de Relatorio de Teste";
```

```
        echo $descreltes >> $STDIR/ConfST;
fi
```

3 Script para criação da sessão de análise de cobertura (qcriasac)

```
#!/bin/sh
#############################################################################
#
# Nome: qcriasac
#
# Entradas: nome da SAC, nome da ST, ferramenta usada e criterio apoiado
#
# Saida: Arquivo da SAC contendo nome do criterio e da ferramenta e as
#         entradas para o IC de teste da SAC
#
#############################################################################

Usage(){
        PROG=`basename $0`
        echo "Use: $PROG SAC ST ferramenta criterio"
        echo "Cria uma Sessao de Analise de Cobertura (SAC)"
        }

if test $# -lt 4
then
        Usage;
        exit;
fi

### Analisa se ST esta cadastrada
```

```
###
STDIR=$LIBDIR/$2;
if !test -d $STDIR
        then    echo "Sessao de teste nao existe!!!!";
                exit 1;
fi

### Analisa se ferramenta e criterio estao cadastrados
###
grep $4 $STDIR/ArqST > $STDIR/saida;
if test -s $STDIR/saida
        then    crit2=`awk '{print $1}' $STDIR/saida`;
                ferr2=`awk '{print $2}' $STDIR/saida`;
                if test "$3" != "$ferr2"
                        then    echo "Ferramenta nao cadastrada para a sessao de teste $2";
                        rm $STDIR/saida;
                                exit 1;
                fi
        else    echo "Criterio nao cadastrado para a sessao de teste $2";
                rm $STDIR/saida;
                exit;
fi

### Cria arquivo de cadastro das SAC
###
echo "$1 $3 $4" >> $STDIR/ArqSAC;
### Cria entradas para os IC de teste
###
```

```
cd $STDIR;
rm saida;
if test ! -d WORK
        then    mkdir WORK;
                mkdir WORK/RCS;
fi
```

4 Salvamento da sessão de análise de cobertura (qsalvasac)

```
#!/bin/sh
##############################################################################
#
# Nome: qsalvasac
#
# Entradas: SAC, ST e motivo da criacao da versao
#
# Saida: Uma versao dos IC de teste da SAC salvos
#
##############################################################################

Usage(){
        PROG=`basename $0`
        echo "Use: $PROG SAC ST motivo"
        echo "Salva uma versao de uma Sessao de Analise de Cobertura (SAC)"
        }

if test $# -lt 3
then
        Usage;
        exit;
```

```
fi

### Analisa se ST esta cadastrada
###
STDIR=$LIBDIR/$2;
if test ! -d $STDIR
        then    echo "Sessao de teste nao existe!!!!";
                exit 1;
fi

### Acerta diretorio de trabalho
###
WORKDIR=$STDIR/WORK;

### Verifica se SAC esta cadastrada e recupera ferramenta e criterio
###
grep $1 $STDIR/ArqSAC > $STDIR/saida;
if test -s $STDIR/saida
        then    ferr=`awk '{print $2}' $STDIR/saida`;
                crit=`awk '{print $3}' $STDIR/saida`;
                rm $STDIR/saida;
        else    echo "SAC nao cadastrada para a sessao de teste $2";
                rm $STDIR/saida;
                exit;
fi

### Salva configuracao dos IC de teste e cria baseline de teste
###
```

```
cd $WORKDIR;
wc -l $CONFDIR/$ferr/$crit/ArqIC > arqaux;
total=`awk '{print $1}' arqaux`;
rm arqaux;
cont=1;
while [ $cont -le $total ]
do
        arqnome=`head -$cont $CONFDIR/$ferr/$crit/ArqIC | tail -1`;
        cont=`expr $cont + 1`;
        ender=`head -$cont $CONFDIR/$ferr/$crit/ArqIC | tail -1`;
        cont=`expr $cont + 1`;
        descr=`head -$cont $CONFDIR/$ferr/$crit/ArqIC | tail -1`;
        cont=`expr $cont + 1`;

        cp $ender/$arqnome .;

        ls -1 $arqnome > arqtemp;
        wc -l arqtemp > arqsize;
        tam=`awk '{print $1}' arqsize`;
        rm -f arqsize;
        ct=1;
        while [ $ct -le $tam ]
        do
                par=`head -$ct arqtemp | tail -1`;
                if test -f RCS/$par,v
                        then    rcs -l $par;
                                ci -q -m"$3" $par;
                        else    ci -q -i -r0.0 -t-"$descr" -m"$3" $par;
                fi
```

```
                ct=`expr $ct + 1`;
        done
        rm -f arqtemp;
done

## Verifica se existem documentos de teste e os salva no IC de teste
##
wc -l $STDIR/ConfST > arqaux;
total=`awk '{print $1}' arqaux`;
rm arqaux;
cont=1;
while [ $cont -le $total ]
do
        arqnome=`head -$cont $STDIR/ConfST | tail -1`;
        cont=`expr $cont + 1`;
        ender=`head -$cont $STDIR/ConfST | tail -1`;
        cont=`expr $cont + 1`;
        descr=`head -$cont $STDIR/ConfST | tail -1`;
        cont=`expr $cont + 1`;

        cp $ender/$arqnome .;

        ls -1 $arqnome > arqtemp;
        wc -l arqtemp > arqsize;
        tam=`awk '{print $1}' arqsize`;
        rm -f arqsize;
        ct=1;
        while [ $ct -le $tam ]
        do
```

```
        par=`head -$ct arqtemp | tail -1`;
        if test -f RCS/$par,v
                then    rcs -l $par;
                        ci -q -m"$3" $par;
                else    ci -q -i -r0.0 -t-"$descr" -m"$3" $par;
        fi
        ct=`expr $ct + 1`;
    done
    rm -f arqtemp;
done
```

5 Congelamento da sessão de análise de cobertura (qcongelasac)

```
#!/bin/sh -x
#############################################################################
#
# Nome: qcongelasac
#
# Entradas: SAC e ST
#
# Saida: Configuracao dos IC de teste da SAC salvos com o nome da SAC
#
#############################################################################

Usage(){
    PROG=`basename $0`
    echo "Use: $PROG SAC ST"
    echo "Cria a baseline de uma Sessao de Analise de Cobertura (SAC)"
    }
```

```
if test $# -lt 2
then
        Usage;
        exit;
fi

### Analisa se ST esta cadastrada
###
STDIR=$LIBDIR/$2;
if test ! -d $STDIR
        then    echo "Sessao de teste nao existe!!!!";
                exit 1;
fi

### Acerta diretorio de trabalho
###
WORKDIR=$STDIR/WORK;

### Verifica se SAC esta cadastrada e recupera ferramenta e criterio
###
grep $1 $STDIR/ArqSAC > $STDIR/saida;
if test -s $STDIR/saida
        then    ferr=`awk '{print $2}' $STDIR/saida`;
                crit=`awk '{print $3}' $STDIR/saida`;
                rm $STDIR/saida;
        else    echo "SAC nao cadastrada para a sessao de teste $2";
                rm $STDIR/saida;
                exit;
fi
```

```
### Salva configuracao dos IC de teste e cria baseline de teste
###

cd $WORKDIR;
wc -l $CONFDIR/$ferr/$crit/ArqIC > arqaux;
total=`awk '{print $1}' arqaux`;
rm arqaux;
cont=1;
while [ $cont -le $total ]
do
        arqnome=`head -$cont $CONFDIR/$ferr/$crit/ArqIC | tail -1`;
        cont=`expr $cont + 1`;
        ender=`head -$cont $CONFDIR/$ferr/$crit/ArqIC | tail -1`;
        cont=`expr $cont + 1`;
        descr=`head -$cont $CONFDIR/$ferr/$crit/ArqIC | tail -1`;
        cont=`expr $cont + 1`;
        cp $ender/$arqnome .;

        ls -1 $arqnome > arqtemp;
        wc -l arqtemp > arqsize;
        tam=`awk '{print $1}' arqsize`;
        rm -f arqsize;
        ct=1;
        while [ $ct -le $tam ]
        do
                par=`head -$ct arqtemp | tail -1`;
                if test -f RCS/$par,v
                        then     rcs -l $par;
```

```
                        ci -q -m"baseline da SAC $1" $par;
                else    ci -q -i -r0.0 -t-"$descr" -m"baseline da SAC $1" $par;
            fi
            ct=`expr $ct + 1`;
        done
        rm -f arqtemp;
        rcs -n"SACbase-$1": RCS/$arqnome,v;
done

## Verifica se existem documentos de teste e os salva na baseline de teste
##

wc -l $STDIR/ConfST > arqaux;
total=`awk '{print $1}' arqaux`;
rm arqaux;
cont=1;
while [ $cont -le $total ]
do
        arqnome=`head -$cont $STDIR/ConfST | tail -1`;
        cont=`expr $cont + 1`;
        ender=`head -$cont $STDIR/ConfST | tail -1`;
        cont=`expr $cont + 1`;
        descr=`head -$cont $STDIR/ConfST | tail -1`;
        cont=`expr $cont + 1`;
        cp $ender/$arqnome .;

        ls -1 $arqnome > arqtemp;
        wc -l arqtemp > arqsize;
        tam=`awk '{print $1}' arqsize`;
```

```
        rm -f arqsize;
        ct=1;
        while [ $ct -le $tam ]
        do
                par=`head -$ct arqtemp | tail -1`;
                if test -f RCS/$par,v
                        then    rcs -l $par;
                                ci -q -m"baseline da SAC $1" $par;
                        else    ci -q -i -r0.0 -t-"$descr" -m"baseline da SAC $1" $par;
                fi
                ct=`expr $ct + 1`;
        done
        rm -f arqtemp;
        rcs -n"SACbase-$1": RCS/$arqnome,v;
done
```

6 Recuperação da sessão de análise de cobertura (qrecuperasac)

```
#!/bin/sh -x
##############################################################################
#
# Nome: qrecuperasac
#
# Entradas: SAC e ST
#
# Saida:
#
##############################################################################

Usage(){
```

```
        PROG=`basename $0`
        echo "Use: $PROG SAC ST"
        echo "Recupera a configuracao congelada da SAC"
        }

if test $# -lt 2
then
        Usage;
        exit;
fi

### Analisa se ST esta cadastrada
###
STDIR=$LIBDIR/$2;
if test ! -d $STDIR
        then    echo "Sessao de teste nao existe!!!!";
                exit 1;
fi

### Acerta diretorio de trabalho
###
WORKDIR=$STDIR/WORK;

### Verifica se SAC esta cadastrada e recupera ferramenta e criterio
###
grep $1 $STDIR/ArqSAC > $STDIR/saida;
if test -s $STDIR/saida
        then    ferr=`awk '{print $2}' $STDIR/saida`;
                crit=`awk '{print $3}' $STDIR/saida`;
```

```
                rm $STDIR/saida;
        else    echo "SAC nao cadastrada para a sessao de teste $2";
                rm $STDIR/saida;
                exit;
fi

### Recupera a baseline de teste e recoloca os arquivos nos locais corretos
###

cd $WORKDIR;
co -r"SACbase-$1" RCS/*;
wc -l $CONFDIR/$ferr/$crit/ArqIC > arqaux;
total=`awk '{print $1}' arqaux`;
rm arqaux;
cont=1;
while [ $cont -le $total ]
do
        arqnome=`head -$cont $CONFDIR/$ferr/$crit/ArqIC | tail -1`;
        cont=`expr $cont + 1`;
        ender=`head -$cont $CONFDIR/$ferr/$crit/ArqIC | tail -1`;
        cp $arqnome $ender;
        chmod 777 $arqnome;
        rm $arqnome;
        cont=`expr $cont + 2`;
done
```

Anexo G - O programa `prime.c` instrumentado pela Poke-tool

```
#define ponta_de_prova(num) if(++printed_nodes % 10) fprintf(path," %2d ",num);\
else fprintf(path," %2d\n",num);

#include <stdio.h>

main()
/*  1 */       {
               FILE * path = fopen("main/path.tes","a");
               static int printed_nodes = 0;
/*  1 */          int                  numero, i;
               ponta_de_prova(1);
/*  1 */          printf("Digite um numero inteiro (tecle 0 para sair): ");
/*  1 */          scanf("%d", &numero);
/*  2 */          while(numero)
/*  3 */          {
                 ponta_de_prova(2);
                 ponta_de_prova(3);
/*  3 */             if(numero % 2 == 0)
/*  4 */             {
                   ponta_de_prova(4);
/*  4 */                printf("%d nao e' um numero primo. \n", numero);
/*  4 */             }
/*  5 */             else
/*  5 */             {
                   ponta_de_prova(5);
```

```
/*  5 */              i = 3;
/*  6 */              while(i <= numero / 2 && numero % i != 0)
/*  7 */              {
                     ponta_de_prova(6);
                     ponta_de_prova(7);
/*  7 */                i++;
/*  7 */              }
                   ponta_de_prova(6);
                   ponta_de_prova(8);
/*  8 */              if(i <= numero / 2)
/*  9 */              {
                     ponta_de_prova(9);
/*  9 */                printf("%d nao e' um numero primo. \n", numero);
/*  9 */              }
/* 10 */               else
/* 10 */               {
                     ponta_de_prova(10);
/* 10 */                 printf("%d e' um numero primo!! \n", numero);
/* 10 */               }
                   ponta_de_prova(11);
/* 11 */             }
                 ponta_de_prova(12);
/* 12 */             printf("Digite outro numero inteiro (tecle 0 para sair): ");
/* 12 */             scanf("%d", &numero);
/* 12 */           }
               ponta_de_prova(2);
               ponta_de_prova(13);
/* 13 */           printf("Ate logo \n");
               fclose(path);
```

```
/* 13 */        }
```