UNIVERSIDADE TECNOLÓGICA FEDERAL DO PARANÁ
CAMPUS CURITIBA

GERÊNCIA DE PESQUISA E PÓS-GRADUAÇÃO

PROGRAMA DE PÓS-GRADUAÇÃO EM ENGENHARIA ELÉTRICA E INFORMÁTICA INDUSTRIAL - CPGEI

JOÃO CADAMURO JUNIOR

DYRETIVA: UM MÉTODO PARA A VERIFICAÇÃO DAS RESTRIÇÕES TEMPORAIS EM SISTEMAS EMBARCADOS

TESE DE DOUTORADO

CURITIBA
OUTUBRO - 2007.

UNIVERSIDADE TECNOLÓGICA FEDERAL DO PARANÁ
Programa de Pós-Graduação em Engenharia Elétrica e Informática Industrial

TESE
apresentada à UTFPR
para a obtenção do título de

DOUTOR EM CIÊNCIAS

por

JOÃO CADAMURO JUNIOR

DYRETIVA: UM MÉTODO PARA A VERIFICAÇÃO DAS RESTRIÇÕES TEMPORAIS EM SISTEMAS EMBARCADOS

Banca Examinadora:

Presidente e Orientador:

Prof. Dr. Douglas P. B. Renaux	UTFPR

Examinadores:

Prof. Dr. Rômulo Silva de Oliveira	UFSC
Prof. Dr. Célio Estevan Moron	UFSCar
Prof. Dr. Paulo Cézar Stadzisz	UTFPR
Prof.ª Dr.ª Keiko Verônica Ono Fonseca	UTFPR

Curitiba, outubro de 2007.

JOÃO CADAMURO JUNIOR

DYRETIVA: UM MÉTODO PARA A VERIFICAÇÃO DAS RESTRIÇÕES TEMPORAIS EM SISTEMAS EMBARCADOS

Tese apresentada ao Programa de Pós-Graduação em Engenharia Elétrica e Informática Industrial da Universidade Tecnológica Federal do Paraná, como requisito parcial para a obtenção do título de "Doutor em Ciências" – Área de Concentração: Informática Industrial.

Orientador: Prof. Dr. Douglas P. B. Renaux

Curitiba

2007

Ficha catalográfica elaborada pela Biblioteca da UTFPR – Campus Curitiba

C121d Cadamuro Junior, João
Dyretiva : um método para a verificação das restrições temporais em sistemas embarcados / João Cadamuro Junior. Curitiba. UTFPR, 2007
XV, 150 f. : il. ; 30 cm

Orientador: Prof. Dr. Douglas P. B. Renaux
Tese (Doutorado) – Universidade Tecnológica Federal do Paraná. Curso de Pós-Graduação em Engenharia Elétrica e Informática Industrial. Curitiba, 2007
Bibliografia: f. 133 – 139

1. Engenharia de sistemas. 2. Sistemas em tempo real. 3. Software. I. Renaux, Douglas P. B., orient. II. Universidade Tecnológica Federal do Paraná. Curso de Pós-Graduação em Engenharia Elétrica e Informática Industrial. III. Título.

CDD. 620.7

AGRADECIMENTOS

A Deus, pela graça da existência e pela oportunidade de viver aqui na Terra nesta era do conhecimento e da evolução tecnológica.

A minha esposa Rosana e a meus filhos Gabriel e Bruno, por graciosamente abrirem mão de minha presença para que este trabalho pudesse ser feito.

A meu pai, que ainda torce por meu sucesso como nos tempos em que eu era apenas um menino.

A minha mãe, que da imensidão da eternidade também acompanha este momento.

Ao orientador Douglas, que dedicou tempo para as muitas correções de rumo que foram necessárias ao longo desta jornada.

Aos gerentes do LACTEC Carlos Zanetti e Welington Desan, que viabilizaram recursos para experiências e participações em conferências que não teriam sido conseguidos de outra maneira.

Ao amigo Cláudio Navarro, por voluntariamente compartilhar seu conhecimento e seu tempo pessoal para auxiliar na construção da ferramenta gráfica de apresentação dos resultados do SoftScope.

Ao colega de pós-graduação Luiz Fernando Copetti, que utilizou sua experiência em projetos de lógica programável para construir o MIMO, o monitor híbrido do SoftScope.

Ao dedicado aluno de graduação Igor Alexandre Modesto, que com inteligência, esforço e superação materializou o pré-instrumentador e o instrumentador do SoftScope.

SUMÁRIO

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE ABREVIATURAS E SIGLAS

ALU	Arithmetic Logic Unit (unidade lógica e aritmética)
ARM	Advanced RISC Machine (máquina RISC avançada)
BCET	Best Case Execution Time (tempo de execução no melhor caso)
CPGEI	Programa de Pós-Graduação em Engenharia Elétrica e Informática Industrial da UTFPR
DMA	Direct Memory Access (acesso direto à memória)
Dyretiva	Dynamic Real-Time Violation Analyzer (analisador de violações dinâmicas de sistemas em tempo real)
ERTS	Embedded Real-Time System (sistema emparcado operando em tempo real)
FIFO	First In, First Out (primeiro a entrar, primeiro a sair)
ICE	In-Circuit Emulator (emulador no circuito)
IDE	Integrated Development Environment (ambiente integrado para o desenvolvimento de *software*)
IEEE	Institute of Electrical and Electronics Engineers (instituto dos engenheiros eletricistas e eletrônicos)
INPI	Instituto Nacional da Propriedade Industrial
IUT	Implementation Under Test (implementação sob teste)
JTAG	Joint Test Action Group (norma IEEE 1149.1)
LED	Light-Emitting Diode (diodo emissor de luz)
LIT	Laboratório de Inovação e Tecnologia em Sistemas Embarcados da UTFPR
LTT	Linux Trace Toolkit (ferramenta de rastreamento do núcleo do sistema operacional Linux)
MIMO	Minimal Invasive Monitor (monitor de intrusão mínima)

MMU	Memory Management Unit (unidade de gerenciamento de memória)
OBCET	Observed Best Case Execution Time (tempo de execução observado no melhor caso)
OCD	On-Chip Debug (depuração no próprio componente)
OTCET	Observed Typical Case Execution Time (tempo de execução observado no caso típico)
OWCET	Observed Worst Case Execution Time (tempo de execução observado no pior caso)
PLEL	Process Level Execution Log (registro de execução a nível de processo)
RAM	Random Access Memory (memória de acesso aleatório)
RISC	Reduced Instruction Set Computer (computador com conjunto reduzido de instruções)
RTOS	Real-Time Operating System (sistema operacional tempo real)
SDRAM	Syncrhonous Dynamic RAM (RAM dinâmica síncrona)
SDR-SDRAM	Single Data Rate SDRAM (SDRAM com taxa de dados simples)
SoC	System on Chip (sistema implementado com um único componente, tipicamente de lógica programável)
SUT	System Under Test (sistema sob teste)
TCET	Typical Case Execution Time (tempo de execução no caso típico)
TLM	Time Line Monitor (monitor de linha de tempo)
TTY	Teletypewriter (terminal teletipo)
UTFPR	Universidade Tecnológica Federal do Paraná
VDS	Visualization and Debugging System (sistema de visualização e depuração)
WCET	Worst Case Execution Time (tempo de execução no pior caso)

RESUMO

O Dyretiva é um método desenvolvido para utilização na fase de testes de sistemas embarcados operando em tempo real e, em especial, na verificação das restrições temporais do sistema. Como a fase de testes situa-se no final do processo de desenvolvimento, quando o *hardware* está disponível e o *software* codificado, a verificação temporal é feita por meio de monitoração do sistema sob teste.

As principais premissas do Dyretiva são considerar a limitação de recursos dos sistemas embarcados e as características intrínsecas dos sistemas em tempo real. O método é definido por uma abordagem de monitoração e por um modelo de falta. A abordagem de monitoração define a interface física e lógica necessárias para observar o sistema sob teste, bem como as estratégias de utilização que permitem otimizar a coleta de dados. O modelo de falta identifica as relações e componentes do sistema onde existe maior probabilidade de encontrar os erros procurados.

Para demonstrar os conceitos do Dyretiva, um conjunto de ferramentas de apoio a aplicação do método foi construído. Este conjunto, chamado de SoftScope, é composto por seis ferramentas: um pré-instrumentador de código, um instrumentador de código, um monitor, um programa de controle do monitor, programas para filtragem e análise dos dados capturados e um programa de visualização dos resultados.

O Dyretiva e o SoftScope são parte integrante do projeto PERF, que está em andamento no LIT (Laboratório de Inovação e Tecnologia em Sistemas Embarcados) da UTFPR (Universidade Tecnológica Federal do Paraná), cujo objetivo é construir um ambiente completo para o desenvolvimento de sistemas embarcados operando em tempo real.

ABSTRACT

The Dyretiva is a method used for verifying the time constraints of embedded real-time systems. The verification is performed by monitoring the embedded software when it is running in an embedded hardware.

The Dyretiva method takes into account the resource constrained nature of embedded systems and the time bounded nature of real-time systems. The method is comprised by a monitoring approach and a fault model. The monitoring approach defines the physical and the logical interfaces used in the observation of the system under test, as well as the strategies used for an optimized trace data collection. The fault model identifies relationships and components of the system under test that are most likely to have time faults.

To demonstrate Dyretiva concepts, a set of support tools called SoftScope has been developed. SoftScope is comprised of a source code pre-instrumentation tool, a source code instrumentation tool, a hybrid monitor, a program for controlling the hybrid monitor, programs for filtering and analyzing trace data, and a graphical presentation tool.

The Dyretiva method and the SoftScope tool set are an integral part of the work-in-progress PERF project, which is under development in the LIT (Laboratory of Embedded Systems Innovation and Technology), at the UTFPR (Federal Technological University of Paraná State). The objective of the PERF project is to build a complete environment suitable for the development of embedded and real-time systems.

1 INTRODUÇÃO

Sistemas computacionais embarcados, ou simplesmente sistemas embarcados, são aqueles que se inserem em equipamentos ou em sistemas sócio-técnicos [Sommerville 07] com o objetivo de controlá-los, ou monitorá-los, ou ainda automatizá-los. Sistemas de automação e controle industriais, sistemas de controle em automóveis, equipamentos de telecomunicações, dispositivos para automação comercial e aparelhos eletro-eletrônicos de uso residencial são exemplos de sistemas embarcados.

Uma parcela significativa dos sistemas embarcados opera sob restrições temporais; estes se denominam sistemas embarcados operando em tempo real, ou simplesmente ERTS (*Embedded Real-Time System*). Nestes sistemas não basta que as respostas produzidas pela computação sejam corretas, elas precisam também ser geradas dentro de limites de tempo bem definidos [Burns 97].

Em algumas áreas de aplicação, um ERTS é classificado como de segurança crítica. Nestes casos, uma falha de lógica ou de temporização no sistema pode causar perdas humanas, econômicas ou ambientais. Equipamentos médicos de suporte à vida, computadores de bordo de aeronaves e sistema digitais de controle de plantas nucleares são exemplos de ERTS de segurança crítica.

Nos sistemas em tempo real, de forma mais enfática do que nos sistemas de computação para processamento de dados, é necessário garantir a qualidade do *software* embarcado. Para os ERTS, características como previsibilidade, confiabilidade, robustez e segurança são obrigatórias. Utilizando os fundamentos da engenharia de *software* [Pressman 95], este nível de qualidade só pode ser alcançado com a aplicação de um conjunto abrangente de testes. Segundo este autor, a atividade de 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. A garantia da qualidade de *software* em ERTS por meio de testes também é ressaltada por Burns [Burns 97] ao afirmar que o uso de métodos formais durante o processo de desenvolvimento não diminui a necessidade de testes, pois estas duas estratégias são complementares e precisam ser utilizadas em conjunto.

Considerando a necessidade de testes em ERTS para garantir a qualidade do *software* embarcado, e reconhecendo que nestes casos existem particularidades que normalmente não existem em outros sistemas computacionais, este trabalho propõe um método e um conjunto de ferramentas para utilização na fase de testes de ERTS, com ênfase na verificação das restrições temporais.

1.1 OBJETIVO

O objetivo deste trabalho de pesquisa é definir um método de verificação das restrições temporais de sistemas embarcados que operam em tempo real. A verificação temporal é uma atividade ligada à garantia da qualidade do *software* de tempo real que acontece durante a fase de testes do sistema. Esta atividade deve certificar-se de que o comportamento temporal do sistema, incluindo as reações aos estímulos vindos do ambiente, esteja de acordo com a especificação temporal.

Para que o método atinja o objetivo definido é necessário que a análise do problema considere as particularidades dos sistemas de computação embarcada e a importância da dimensão temporal nos sistemas em tempo real.

Quando comparada à computação com o objetivo de processamento de dados, a computação embarcada possui características próprias, como recursos restritos e dimensionados para atender somente a aplicação a que se destinam. Um exemplo que ilustra esta questão é a capacidade de processamento e a quantidade de memória volátil. Em equipamentos embarcados utilizados em telecomunicações é possível encontrar módulos computacionais cujos processadores operam em freqüências de relógio acima de gigahertz, que possuem dezenas de megabytes de memória, em configuração semelhante aos servidores de rede utilizados no processamento de dados. No outro extremo, existem aplicações em automóveis e eletrodomésticos cujos processadores operam em freqüências de relógio de dezenas de megahertz, e apenas algumas centenas de bytes de memória. Ao propor uma metodologia que se aplica a sistemas embarcados, é preciso considerar a variedade que o tema envolve, procurando atingir o maior número possível de aplicações.

Considerar a dimensão temporal durante a fase de testes é um desafio porque formas comuns de obter informações sobre o comportamento dinâmico e temporal de um sistema de computação com o objetivo de processamento de dados não podem ser utilizadas em sistemas com restrições temporais. Um exemplo de técnica que não se aplicada em sistemas embarcados com restrições temporais é o registro da ocorrência de eventos em arquivos, pois

o tempo gasto com a anotação do evento e a gravação do arquivo alteram o comportamento temporal, além de muitos sistemas embarcados não possuírem memória de massa. Outro exemplo é a utilização de depuradores de código (*debuggers*) para encontrar erros, pois os erros que possuem origem no descumprimento de restrições temporais não podem ser observados através da paralisação do programa, o que provocaria uma mudança no comportamento temporal do sistema.

A definição do método Dyretiva de verificação das restrições temporais em ERTS é um trabalho de pesquisa incluído no contexto do projeto PERF [Renaux 99]. O objetivo do PERF é construir um ambiente completo de verificação de ERTS.

1.2 JUSTIFICATIVA E RELEVÂNCIA

Em 2002, cerca de 98% dos microprocessadores comercializados no mundo foram utilizados em sistemas embarcados [Turley 03], o que mostra a importância desta área na atualidade.

Uma grande parte do esforço de desenvolvimento de um sistema computacional é despendida nas etapas finais do processo, quando as atividades de integração e testes são levadas a efeito. Estatísticas mostram que estas atividades consomem em torno de 40% do tempo total de desenvolvimento, considerando como paradigma da engenharia de *software* o ciclo de vida clássico, também chamado de modelo cascata [Pressman 95]. No caso extremo das atividades de testes de sistemas dos quais dependem vidas humanas, como no caso dos sistemas em tempo real de segurança crítica, o esforço com testes pode consumir de três a cinco vezes mais do que todas as outras etapas do processo de desenvolvimento de *software* juntas [Pressman 95]. No caso de ERTS, uma parte considerável deste esforço de testes é despendida na verificação das restrições temporais, o que implica certificar-se de que o sistema é capaz de tratar adequadamente os diversos estímulos que podem vir do ambiente dentro dos limites temporais estabelecidos para que a segurança, a previsibilidade e a robustez do sistema sejam garantidas.

Considerando o esforço necessário para testar a funcionalidade e as restrições temporais de um sistema em tempo real, métodos e ferramentas que possam ser aplicados durante a fase de testes destes sistemas são importantes por três motivos principais: dão visibilidade à dinâmica do sistema, reduzem o tempo de teste e aumentam a confiabilidade do sistema.

Obter visibilidade sobre a dinâmica do sistema é importante porque as falhas de temporização não acontecem em virtude de um erro de lógica do programa, mas sim devido a um conjunto de fatores que não permitem que o resultado do processamento lógico seja gerado dentro do limite de tempo especificado. Como este tipo de problema não ocorre em computação com o objetivo de processamento de dados, métodos e ferramentas com este propósito não são comuns. Além disso, é necessário que a monitoração do ERTS seja feita sem modificar o comportamento temporal do sistema, evitando assim incorrer no que se conhece como Princípio da Incerteza de Heisenberg aplicado ao *software* [Laplante 90].

A redução no tempo de realização dos testes é importante porque se sabe que esta fase consome uma parte significativa do tempo total de desenvolvimento de um ERTS. Uma redução de tempo, neste caso, pode ser alcançada pela automatização dos testes temporais, tornando-os mais eficientes e menos propícios a erros humanos. Entretanto, o método e as ferramentas devem garantir que esta redução seja alcançada sem perda da qualidade do *software*.

Finalmente, a confiabilidade é uma das características esperadas de um ERTS, e pode ser definida como uma avaliação subjetiva da probabilidade de existirem erros ainda não descobertos no sistema. Um aumento de confiabilidade é conseguido pela garantia de que os testes realizados são suficientemente abrangentes, e pela eliminação dos erros descobertos durante os testes.

1.3 CONTRIBUIÇÕES

Os requisitos e motivações deste trabalho basearam-se nas necessidades e dificuldades encontradas no desenvolvimento de sistemas embarcados operando em tempo real realizados no LIT (Laboratório de Inovação e Tecnologia em Sistemas Embarcados) da UTFPR (Universidade Tecnológica Federal do Paraná), ao longo de mais de dez anos de trabalho na área. Durante este tempo, alguns métodos e ferramentas foram estudados e experimentados na realização de testes e verificação temporal. Entretanto, todos eles possuem alguma desvantagem intrínseca que torna sua utilização limitada. Uns dependem de compiladores modificados para garantir propriedades temporais, enquanto outros dependem de segredos industriais para coletar informações sobre a execução do sistema. Alguns só se aplicam a um modelo de processador em particular, enquanto outros estão ligados a um sistema operacional específico. Alguns exigem que o sistema sob teste possua muitos recursos disponíveis, enquanto outros não atentam para as necessidades dos ERTS de

segurança crítica. Uma parte deste universo de métodos e ferramentas é apresentado na Seção 4.4.

Com base nesta experiência, a contribuição deste trabalho é a definição de um método de verificação das restrições temporais de sistemas embarcados operando em tempo real que considera tanto as características dos sistemas embarcados quanto as necessidades dos sistemas em tempo real, sobrepujando as limitações encontradas nos métodos e soluções estudados na literatura ou experimentados na prática. O resultado é um método que não depende de um compilador ou ambiente de desenvolvimento em particular, utiliza tecnologias abertas para observar o sistema sob teste, é independente da arquitetura do processador e do sistema operacional utilizados, exige poucos recursos do sistema sob teste, e adere às necessidades dos sistemas em tempo real de segurança crítica.

1.4 ESTRUTURA

Este trabalho está organizado em oito capítulos, uma seção de referências e dois anexos. O Capítulo 2 apresenta tópicos em engenharia de *software* para sistemas em tempo real, cujo objetivo é o de contextualizar o trabalho. O Capítulo 3 aprofunda um dos assuntos tratados no Capítulo 2 e que possui particular importância para o método Dyretiva: a medição do tempo de execução de programas. O Capítulo 4 descreve os trabalhos que foram encontrados na literatura e que estão relacionados com a verificação temporal de sistemas embarcados. O Capítulo 5 discute os requisitos, processos e modelos que compõe o método Dyretiva. O Capítulo 6 descreve o SoftScope, um conjunto de ferramentas construído para permitir a aplicação do método Dyretiva. No Capítulo 7 são mostrados resultados da utilização do método Dyretiva em casos de teste reais. No Capítulo 8 estão enumeradas as conclusões do trabalho, bem como algumas sugestões para trabalhos futuros. A seção seguinte contém as referências utilizadas na elaboração do trabalho. O Anexo I apresenta um histórico e algumas características técnicas do PET#, um núcleo de tempo real desenvolvido para realizar os experimentos de validação do Dyretiva. Por fim, o Anexo II traz informações sobre o protocolo de comunicação utilizado pelo programa de controle do monitor, uma das ferramentas do SoftScope.

2 TÓPICOS EM ENGENHARIA DE SOFTWARE PARA SISTEMAS EM TEMPO REAL

A área de Engenharia de *Software* necessita de abordagens de engenharia para que os resultados produzidos tenham a funcionalidade e a qualidade almejadas, e que sejam alcançados dentro de um período de tempo pré-estabelecido. Assim como outras abordagens de engenharia, o desenvolvimento de *software* precisa de métodos, ferramentas e processos. Os métodos introduzem formas específicas de realizar tarefas. As ferramentas proporcionam apoio na aplicação dos métodos. Os processos definem a seqüência em que os diversos métodos serão aplicados [Pressman 95].

O desenvolvimento de *software* para sistemas em tempo real possui desafios adicionais, além daqueles já existentes nos sistemas de computação para processamento de dados, pois os aspectos temporais tornam-se tão importantes quanto os aspectos funcionais. Além disso, sistemas embarcados possuem recursos limitados em um ou mais dos seguintes aspectos: espaço físico, consumo, memória, capacidade de processamento, quantidade e velocidade dos canais de comunicação, entre outros. Deste modo, métodos, ferramentas e processos utilizados no desenvolvimento de ERTS devem ser adaptados as suas condições peculiares, contemplando tanto a importância dos aspectos temporais quanto os recursos restritos.

No contexto do desenvolvimento de *software* para sistemas em tempo real, as seções a seguir apresentam três tópicos relacionados ao assunto que possuem estreita ligação com este trabalho. O primeiro tópico aborda um processo de desenvolvimento utilizado para sistemas em tempo real. O segundo tópico apresenta conceitos de teste de *software*. O terceiro tópico mostra os métodos existentes para determinar o tempo de execução de um programa, atividade necessária para verificar se os requisitos temporais do sistema são cumpridos. Após a apresentação destes três tópicos é introduzido o conceito do ambiente PERF, que é um projeto do LIT cujo objetivo é criar um ambiente voltado ao desenvolvimento e verificação de ERTS. O método Dyretiva está incluído no contexto do projeto PERF.

2.1 PROCESSO DE DESENVOLVIMENTO

O processo de desenvolvimento de ERTS utilizado como referência neste trabalho é o adotado pelo LIT, que combina paradigmas da engenharia de *software*, como o ciclo de vida clássico e a prototipação [Pressman 95], e que considera também a necessidade de desenvolvimento de *hardware*, conforme ilustrado na Figura 1.

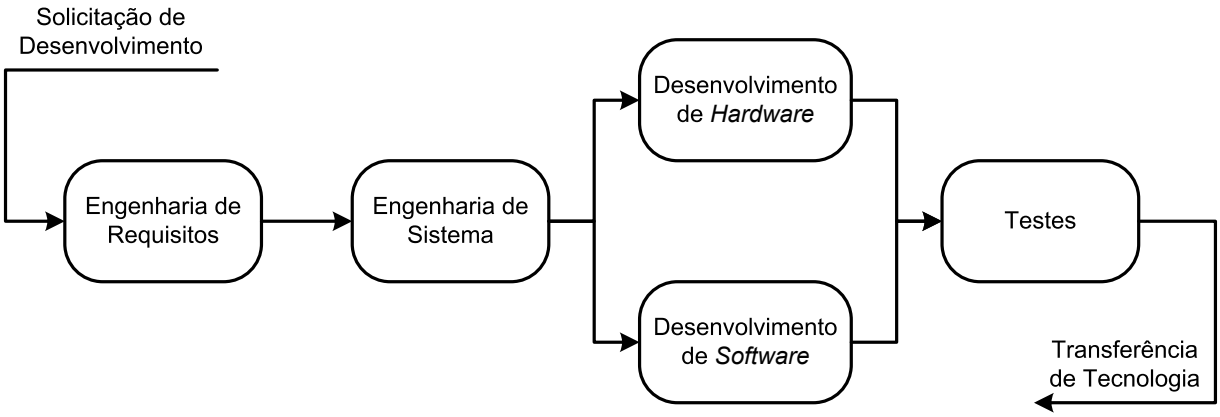


Figura 1: Processo de desenvolvimento de ERTS utilizado no LIT.
Fonte: [Renaux 01]

De acordo com o processo adotado no LIT, um novo desenvolvimento inicia-se por solicitação de um cliente. A fase inicial do processo é a Engenharia de Requisitos, cujo resultado é o documento de especificação do sistema. Durante esta fase um modelo do sistema é construído levando em consideração questões funcionais, temporais e de desempenho. Diferentes formas de modelagem podem ser utilizadas, dentre as quais podemos citar: máquinas de estado temporizadas, Statecharts [Harel 87], diferentes formas de lógica temporal [Sowmya 98] [Bellini 00] [Mattolini 01] [Bellini 01] [Bellini 06], autômatos temporais [Kaynar 03] [Gebremichael 05] e UML [Douglass 98] [Apvrille 04]. A modelagem direciona a solução adotada para o atendimento das especificações do cliente e, quando associada a ferramentas de apoio, permite a geração de um sistema correto por construção, onde cada parte do sistema pode ser mapeada a um requisito do cliente. O documento de especificação gerado pela Engenharia de Requisitos deve conter pelo menos cinco tópicos: a informação disponível a respeito do domínio do problema, a especificação funcional, a especificação temporal, a especificação das interfaces físicas e lógicas com o ambiente externo e o modelo do sistema.

A fase seguinte no processo de desenvolvimento é a Engenharia de Sistema, cujo objetivo é definir as funcionalidades que serão implementadas em *hardware* e as que serão implementadas em *software*. A construção de protótipos durante esta fase auxilia na verificação de questões relativas à funcionalidade e ao desempenho, ajudando no processo de tomada de decisão. A fase de Engenharia de Sistema gera como resultado dois documentos: a especificação de *hardware* e a especificação de *software* do sistema.

Uma vez concluídas as especificações de *hardware* e de *software*, o desenvolvimento de ambos pode seguir paralelo, conforme ilustrado na Figura 1. A implementação do *hardware* é concluída quando todos os componentes são exercitados e encontram-se funcionando de acordo com o que é esperado deles. A implementação do *software* é concluída quando todas as funções, métodos, classes, módulos e componentes estão codificados, e cada unidade que compõe o sistema foi testada individualmente.

Com o *software* codificado e o *hardware* disponível, inicia-se a fase de testes, que pode ser dividida em duas partes principais: os testes de integração e o teste de sistema. Os testes de integração verificam que unidades de *hardware* e de *software* projetadas para operarem em conjunto estejam funcionando adequadamente. O teste de sistema verifica que o sistema como um todo atende as especificações estabelecidas no início do projeto. Ao final da fase de testes o produto é entregue ao cliente e a tecnologia é transferida para permitir a produção e a manutenção.

Um dos objetivos do processo de desenvolvimento é garantir a qualidade do produto gerado. Para isto, a preocupação com qualidade deve permear todas as etapas do processo, iniciando com os métodos e ferramentas de análise dos requisitos e indo até o controle das versões e da documentação do produto. Apesar da qualidade ser um resultado esperado, uma avaliação efetiva desta qualidade só pode ser feita durante a fase de testes. A atividade de teste de *software*, quando ligada à qualidade do *software*, é parte de um tema mais amplo chamado de verificação e validação. A verificação é definida como a garantia de que cada parte do *software* foi construída de acordo com a sua especificação, e que a especificação atende aos requisitos. Já a validação é uma atividade diferente que visa garantir que os requisitos do sistema são consistentes, e que estão de acordo com as exigências do cliente [Pressman 95].

A próxima seção apresenta definições e conceitos de teste de *software*, uma vez que esta atividade é de vital importância na qualidade do *software*, que por sua vez é uma exigência no *software* feito para ERTS.

2.2 TESTE DE SOFTWARE

Esta seção apresenta uma visão do teste como uma das partes do processo de desenvolvimento, e que também está ligada com a verificação do sistema. A apresentação está subdividida em duas partes: a primeira composta por conceitos de teste que se aplicam a engenharia de *software* como um todo, e a segunda com considerações sobre testes de sistemas em tempo real.

2.2.1 CONCEITOS FUNDAMENTAIS DE TESTE

Esta seção apresenta definições, termos e conceitos relacionados ao teste de *software* que foram retirados do trabalho de Binder [Binder 01], a menos que indicado explicitamente ao contrário.

Teste de *software* pode ser definido como a atividade de executar um programa com o objetivo de descobrir uma falha, utilizando para isto um determinado conjunto de entradas e tendo o programa em um estado conhecido. Um teste bem-sucedido revela uma falha ainda não descoberta [Myers 89].

Uma **falha** (*failure*) é uma manifesta inabilidade do sistema, ou de um de seus componentes, em realizar uma função requerida dentro dos limites estabelecidos. Ela pode ser evidenciada pela geração de uma saída incorreta, ou então pelo uso de tempo ou de espaço fora dos limites. Uma **falta** (*fault*) de *software* refere-se a uma parte do sistema que está codificada erroneamente ou que está ausente. Quando o *software* defeituoso é executado, uma falha pode ser gerada. Um **erro** é uma discrepância entre a condição ou o valor observado no sistema e àquele considerado correto pela especificação ou pela teoria. Um erro é a manifestação de uma falta no sistema [Nelson 90]. Para exemplificar, considere-se um sistema reativo implementado por uma máquina de estados. Uma falha pode ser caracterizada pelo envio de um estímulo que não produz no sistema a resposta esperada. O erro pode ser caracterizado pela entrada do sistema em um estado proibido em conseqüência do estímulo enviado. Já a falta é a execução do código defeituoso que permitiu ao sistema entrar no estado proibido. Erros, faltas e falhas não ocorrem necessariamente em uma relação de um para um. Por exemplo, várias falhas diferentes podem ser causadas pela mesma falta.

Um **componente** de *software* é qualquer parte do *software* que possa ser observada no ambiente de desenvolvimento. Funções, métodos, classes, objetos, módulos, e tarefas são

exemplos de componentes. O código sendo submetido a testes é chamado de IUT (*Implementation Under Test*).

Escopo de um teste é o conjunto dos componentes de *software* que o teste é capaz de exercitar.

O escopo de um **teste de unidade** abrange a uma parte pequena do programa: uma função, um método, um módulo ou um pequeno conjunto destes.

O escopo de um **teste de integração** compreende um sistema ou subsistema completo, o que envolve tanto unidades de *software* quanto de *hardware*. As unidades envolvidas devem cooperar para atender a algum requisito de projeto. Testes de integração exercitam as interfaces entre as unidades do escopo para demonstrar que elas operam em conjunto.

O escopo de um **teste de sistema** compreende uma aplicação completa e integrada. Este teste focaliza-se nas características observadas apenas no sistema como um todo. O escopo do teste de sistema pode ser classificado pelo tipo de conformidade que busca: funcional (entrada/saída), desempenho (tempo de resposta e utilização de recursos), carga ou estresse (resposta em condições de carga máxima e de sobrecarga).

Um teste é **direcionado à faltas** quando seu objetivo é revelar faltas através da observação de falhas. Um teste é **direcionado à conformidade** quando seu objetivo é demonstrar conformidade com uma funcionalidade requerida. Estes objetivos não são mutuamente exclusivos, mas normalmente os testes de unidade e de integração são direcionados à faltas, e o teste de sistema é direcionado à conformidade.

Confiança é a avaliação subjetiva da probabilidade de existirem erros ainda não descobertos em uma determinada implementação. Quando uma implementação é submetida a um conjunto abrangente de testes e é aprovada, a confiança no sistema aumenta.

Um **caso de teste** especifica o estado inicial de uma IUT e do ambiente, as entradas de teste que devem ser utilizadas e os resultados esperados. Por resultados esperados entende-se o que a IUT deve produzir a partir das entradas apresentadas. A especificação do caso de teste pode incluir mensagens geradas pela IUT, exceções, valores de retorno e o estado final da IUT e do ambiente. Um **oráculo** é um meio de produzir os resultados esperados para cada caso de teste. Alguns oráculos automatizados também podem fazem uma avaliação simples, do tipo aprovado ou reprovado, do resultado de um teste.

Um **domínio** de testes é um conjunto de valores de entradas e estados que podem ser utilizados em uma IUT. Um **ponto de teste** é um valor específico para as entradas e

variáveis de estado de um programa, e é escolhido em função do domínio. Um ponto de teste pode ser utilizado em muitos casos de teste.

Um **conjunto de testes** (*test suite*) é uma coletânea de casos de teste tipicamente relacionados com um objetivo de teste ou com uma dependência específica da implementação. Um **plano de teste** é um documento preparado para ser utilizado pelos testadores que explica a abordagem a ser utilizada nos testes: o plano de trabalho, os procedimentos gerais, uma explicação do projeto, e assim por diante.

O **término anormal** é termo utilizado para descrever uma falha cujo resultado é a terminação abrupta do programa. **Omissão** refere-se a um requisito do programa que não está presente na implementação. **Surpresa** refere-se a uma funcionalidade encontrada durante os testes que não pode ser mapeada em nenhum requisito do sistema.

Depuração (*debugging*) é o trabalho necessário para diagnosticar e corrigir um erro ou uma falta. Ele ocorre após a observação de uma falha, ou então quando o desenvolvedor identifica uma falta por inspeção ou por análise automática do código.

Como o número de testes possíveis em um sistema real é praticamente infinito, uma abordagem de testes realística é baseada em um **modelo de falta**. Este modelo busca identificar as relações e componentes do SUT onde se presume haver maior probabilidade de encontrar faltas. Esta presunção pode basear-se em experiência, senso comum, análise ou suspeita. Um **caso de teste interessante** é aquele que tem boas chances de revelar uma falta.

Uma **estratégia de teste** é um algoritmo ou heurística utilizada para criar casos de teste a partir de uma representação, implementação ou modelo de teste. Um **modelo de teste** representa as relações entre os elementos de uma representação ou implementação, e é baseado no modelo de falta. O **projeto de teste** (*test design*) está relacionado com três problemas: a identificação de pontos de teste interessantes, a colocação destes pontos de teste em uma seqüência de teste, e a definição dos resultados esperados para cada ponto de teste na seqüência. **Efetividade de teste** é a habilidade relativa de uma estratégia de teste em encontrar um erro ou uma falta. **Eficiência de teste** é o custo relativo de se encontrar um erro ou uma falta.

Estratégias para o projeto de teste podem ser baseadas em responsabilidades, baseadas em implementação, híbridas ou baseada em faltas. **Projeto de teste baseado em responsabilidades** usa as responsabilidades especificadas ou esperadas de uma unidade, subsistema ou sistema para projetar os testes. Este tipo de projeto de teste também é chamado de “orientado à especificação”, “comportamental”, “funcional” ou de “caixa preta”.

Projeto de teste baseado em implementação utiliza a análise do código fonte para desenvolver os casos de teste. Este tipo de projeto também é chamado de "estrutural" ou de "caixa branca".

Uma abordagem de teste é dita **formal** se sua capacidade de revelar faltas é baseada em análise matemática para selecionar os casos de teste, ou então **heurística**, se baseada na experiência de um especialista.

Projeto de teste híbrido combina as duas estratégias anteriores, e por isso é às vezes chamada de abordagem de "caixa cinza".

A última estratégia de projeto de testes, o **teste baseado em faltas**, utiliza mutações do código original para introduzir faltas, e então verificar se estas faltas são reveladas pelo conjunto de testes proposto.

Um **conjunto de teste exaustivo** requer que todos os valores e seqüências de entrada sejam aplicados ao sistema quando este se encontra e cada estado alcançável do espaço de estados, exercitando assim todos os caminhos possíveis. Esta abordagem resulta em um número muito grande de casos de teste, mesmo para programas triviais. Testes exaustivos são impossíveis de serem aplicados na prática. Logo, teste de *software* refere-se, necessariamente, a um pequeno subconjunto do conjunto de testes exaustivos.

A completude de um conjunto de teste, com respeito a um projeto de caso de teste, é medida pela **cobertura**, que por sua vez é definida como o percentual de componentes de *software* alcançados por uma estratégia e exercitados por um determinado conjunto de teste.

2.2.2 TESTE DE SISTEMAS EMBARCADOS OPERANDO EM TEMPO REAL

Nos ERTS o teste de *software* é uma situação mais complexa do que a do teste de programas de processamento de dados porque existe acoplamento ao mundo exterior por meio de canais de comunicação, sensores e atuadores. Esta característica torna o SUT dependente do ambiente onde ele está inserido e da sua plataforma de *hardware*. Além disso, os ERTS normalmente são implementados com programação concorrente, isolando as funcionalidades do sistema em tarefas diferentes e facilitando a manutenção do código. Entretanto a programação concorrente introduz dificuldades adicionais na fase de testes, pois além dos problemas pertinentes aos programas seqüenciais, existem ainda questões relacionadas à concorrência, tais como: chaveamentos entre tarefas, preempções, interrupções e acessos a regiões críticas [Thane 00]. Em um cenário como este, uma possível

macro-estratégia pode dividir o teste em três etapas: o teste de tarefas individuais, o teste comportamental e o teste de comunicação e sincronização entre tarefas [Pressman 95].

O teste de uma tarefa individual é realizado sem considerar a interação com outras tarefas, sendo capaz de revelar erros de lógica, de função e de algoritmos. No teste comportamental cada um dos eventos externos tratados pela tarefa é gerado separadamente para verificação do comportamento. Assim que os eventos são testados em separado, seqüências de eventos são geradas para a tarefa para verificar outros possíveis erros de comportamento advindos do tratamento de eventos em seqüência. Após a correção dos erros de lógica e de comportamento em cada tarefa, a atividade de teste desvia-se para os erros temporais. Neste teste tarefas assíncronas são estimuladas com diferentes taxas de dados e cargas de processamento para determinar se ocorrerão erros de sincronização entre tarefas. Além disso, as filas de mensagens que interligam diferentes tarefas são exercitadas para identificar possíveis situações de perda de mensagens.

Além de uma estratégia de teste diferenciada, o trabalho de diagnóstico da origem de uma falha em um ERTS não pode seguir os mesmos padrões utilizados pelos programas de processamento de dados. O uso de depuradores de código (*debuggers*), com a inserção de pontos de parada (*breakpoints*) ou de observação (*watchpoints*), bem como a execução passo-a-passo, alteram o comportamento temporal, provocando resultados diferentes dos esperados [Thane 00].

Outra questão que merece atenção durante o teste de ERTS concorrentes é a possibilidade do surgimento de falhas intermitentes. Estas falhas têm duas características principais. A primeira é a difícil reprodução. A segunda é que, mesmo quando é possível reproduzir a falha, tentativas de observá-la por meio de depuradores ou pelo envio de mensagens para um terminal fazem com que ela não se manifeste mais. A origem deste problema é um fenômeno conhecido como efeito sonda (*probe effect*) [Gait 86], que ocorre quando a temporização relativa dos eventos do sistema é modificada em virtude da observação [Thane 00]. Especialistas têm apontado as ferramentas de identificação de falhas intermitentes, relacionadas com a concorrência ou com a temporização, de vital importância na garantia da qualidade de *software* [Havelund 03].

2.3 DETERMINAÇÃO DO TEMPO DE EXECUÇÃO

Nos ERTS a qualidade do *software* está ligada tanto com a estratégia e a abrangência do conjunto de testes quanto com o cumprimento dos requisitos temporais. Entretanto, para verificar se os requisitos temporais estão sendo cumpridos é necessário determinar os tempos de execução das tarefas, serviços e funções do programa. Existem basicamente dois métodos para determinar tempos de execução de programas: estáticos e dinâmicos. Os métodos estáticos baseiam-se em modelos computacionais da arquitetura utilizada, de forma que com estes modelos é possível estimar os tempos de execução das diferentes partes que compõe o programa. Os métodos dinâmicos, por outro lado, baseiam-se na medição do tempo de execução do programa, observando-se o sistema durante seu funcionamento normal. Nas seções a seguir estes métodos são detalhados.

2.3.1 MÉTODOS ESTÁTICOS

Os métodos estáticos estimam o tempo de execução de um programa, e em especial do pior caso de tempo de execução ou WCET (*Worst Case Execution Time*), sem executá-lo na plataforma alvo. É pelo valor estimado do WCET que um método estático garante que um ERTS cumpre seus requisitos temporais. O WCET estimado deve ser sempre maior ou igual ao WCET real [Góes 01b].

Estimativas do tempo de execução podem ser baseadas na análise do código fonte ou na análise do código de máquina (*assembly*) do programa. Esta última abordagem é a mais usada nos casos onde a estimativa deve ser precisa, pois ela é independente do compilador ou mesmo das opções de otimização utilizadas na compilação.

Para realizar estimação baseada no código de máquina é preciso resolver dois problemas principais: a determinação do tempo gasto por cada uma das instruções do programa e a determinação dos possíveis caminhos seguidos pelo programa. Além disso, a determinação do tempo gasto por uma instrução é uma tarefa que exige conhecimento da arquitetura do processador em questão e da plataforma de *hardware* onde o código será executado.

O conhecimento da arquitetura do processador é essencial porque existem diversos fatores construtivos que influenciam no tempo de execução de uma instrução, dentre os quais podemos destacar o *pipeline*. Atualmente o *pipeline* é considerado técnica chave para aumentar o desempenho dos processadores [Hennessy 96], e baseia-se na execução simultânea de várias instruções do programa, com cada instrução executando em um estágio

diferente. O nome da técnica sugere uma analogia com as linhas de montagem de veículos. Com o uso do *pipeline* o tempo médio de execução das instruções diminui, porém um componente não determinístico no tempo de execução de cada instrução é inserido. Este componente não-determinístico é resultado da possível interdependência que possa existir entre as instruções vizinhas de um programa, exatamente como a velocidade de produção de um carro em uma linha de montagem depende da velocidade dos outros carros que estão na mesma linha. Em virtude disto, algumas pesquisas nesta área tentam modelar analiticamente o *pipeline* [Healy 95] [Linhares 01] para realizar estimativas.

Além do *pipeline*, os processadores modernos ainda incorporam outros recursos que visam acelerar o tempo médio de execução das instruções, como: *pipeline* avançado, múltiplas unidades de execução, filas de busca antecipada (*prefetch queues*) e memórias *cache* de instruções e de dados. Estes recursos, apesar de cumprirem seu objetivo, fazem da determinação do tempo de execução de uma instrução em particular uma tarefa complexa, dada a elaborada dependência de contexto.

A plataforma de *hardware* na qual um programa será executado é outro fator que influencia na determinação do tempo de execução. As principais influências do *hardware* são: tempo de acesso das memórias, ciclos de *refresh* em memórias dinâmicas, ciclos de DMA (*Direct Memory Access*) e interrupções. Todos estes fatores influenciam na precisão da análise de quanto tempo durará a execução de um determinado trecho do programa. Neste contexto, o tempo de acesso às memórias e os ciclos de *refresh* são fatores relativamente fáceis de serem modelados. O tempo de acesso às memórias causa um atraso constante na busca de cada instrução, e na leitura ou escrita de cada dado. Já os ciclos de *refresh* podem ser modelados como uma tarefa periódica e não-preemptiva de período conhecido que torna a memória do sistema indisponível durante um curto espaço de tempo. No entanto, a interferência de ciclos de DMA e interrupções são de modelagem mais difícil, pois exige conhecimento prévio da influência de todo o restante do sistema no trecho do programa cujo tempo de execução deseja-se estimar, informação esta que pode não estar disponível em um dado estágio do desenvolvimento.

Determinar os possíveis caminhos de execução de um programa é outro problema que precisa ser resolvido pelos métodos estáticos. Segundo Puschner e Koza [Puschner 89], existem condições que precisam ser satisfeitas para que um programa tenha seu tempo de execução calculado. Estas condições, aplicadas no contexto das linguagens de programação de alto nível, encontram-se nos tópicos abaixo.

- Não deve haver rotinas recursivas, pois a profundidade da recursão e o espaço necessário em pilha para a execução não podem ser determinados estaticamente. Além disso, toda e qualquer rotina recursiva pode ser substituída por outra não-recursiva [Darlington 78].
- Não deve haver ponteiros para funções, pois eles impossibilitam a determinação do fluxo de controle antes da execução. Todas as chamadas de função devem ser explícitas.
- Instruções que desestruturam o programa, como o *goto*, devem ser evitadas sob pena de dificultar ou até mesmo impossibilitar a identificação das estruturas de repetição do programa.
- Cada laço do programa deve ter seu número máximo de iterações determinado pelo programador a priori.

Se estes requisitos forem satisfeitos, então os caminhos seguidos durante a execução do programa podem ser determinados estaticamente. Este tempo é obtido somando-se os tempos gastos na execução de cada uma das instruções que compõe um caminho do programa.

Além do cálculo do tempo de execução de um programa, os métodos estáticos também podem ser utilizados para outros propósitos. Engblom listou alguns destes propósitos [Engblom 97], que são apresentados nos tópicos abaixo.

- Servir de parâmetro para definir a política de escalonamento das tarefas do sistema, pois a maioria dos algoritmos de escalonamento precisa conhecer o tempo de execução de cada tarefa.
- Identificar as partes do programa que estão consumindo mais tempo de processamento. Esta análise pode auxiliar na identificação de algoritmos que precisam ser melhorados, em otimizações de código que precisam ser efetuadas, e como auxílio para decidir implementar uma determinada funcionalidade em *hardware*, ao invés de em *software*.
- Comparar a eficiência de dois algoritmos diferentes, ou então de duas implementações diferentes de um mesmo algoritmo, selecionando a solução que for mais eficiente.
- Dimensionar os recursos do sistema, tais como processador, freqüência de operação e tempo de acesso das memórias.

Em resumo, os métodos estáticos são capazes de calcular o tempo de execução de um programa, e em especial do WCET, sem executá-lo na plataforma alvo. Através da determinação do tempo de execução de cada instrução que compõe o programa, é possível determinar o tempo de funções e tarefas. Com estes dados pode-se, além de verificar se o sistema atende as restrições temporais, obter informações sobre o desempenho de diferentes implementações de algoritmos, e sobre a taxa de utilização do processador. Como todas as informações são obtidas a partir de cálculos e modelos, os métodos estáticos podem ser utilizados em sistemas que ainda não existem fisicamente.

2.3.2 MÉTODOS DINÂMICOS

Na seção anterior foram apresentados os métodos estáticos de determinação do tempo de execução, sendo que dentre as características destes métodos podem ser ressaltadas a complexidade de realizar estimativas precisas e o pessimismo inerente. A complexidade é resultante dos fatores envolvidos, tanto relativos à arquitetura do processador, como relativos à plataforma de *hardware* do usuário. Se apenas alguns aspectos forem levados em consideração, a estimativa é imprecisa. Se todos os aspectos forem considerados, é muito complexa. Além disso, mesmo em uma estimativa precisa o cálculo do WCET precisa ser pessimista em razão de sua obrigação de ser maior ou igual ao pior caso real. No entanto, a concatenação de todos os piores casos – como *pipeline* vazio, fila de *prefetch* vazia, memória *cache* vazia ou desabilitada, ciclo de *refresh* em memória dinâmica, e interrupção crítica ocorrendo simultaneamente durante a execução do trecho de código a ter seu tempo estimado – faz o cálculo ser tão pessimista que tende a levar o projeto ao super-dimensionamento. Para não incorrer em tais problemas, métodos dinâmicos precisam ser considerados para complementar os estáticos.

O objetivo dos métodos dinâmicos é medir o tempo de execução diretamente no SUT, o que acontece pela monitoração do programa. Os monitores que coletam os dados da execução no SUT podem ser classificados em três categorias [Stunkel 91][Schroeder 95][Mahrenholz 01]:

- Monitores baseados em *hardware*.
- Monitores baseados em *software*.
- Monitores híbridos.

O Capítulo 3 é dedicado inteiramente à medição de tempos de execução, onde cada uma destas três categorias será apresentada em detalhe.

2.4 O AMBIENTE PERF

O PERF é um projeto de longa duração do LIT que visa construir um ambiente completo para desenvolvimento e verificação de sistemas em tempo real [Renaux 99]. Uma das contribuições relevantes do PERF é o uso de métodos estáticos e dinâmicos de maneira integrada e complementar na busca de uma solução que acelere e dê confiabilidade ao desenvolvimento de sistemas embarcados operando em tempo real.

O PERF foi construído para ser um ambiente integrado de desenvolvimento que agrega as tarefas de gerenciamento de projetos, edição de código fonte, compilação, depuração lógica e temporal. Através do gerente de projetos o usuário agrega todo o seu código fonte e também dispara ferramentas externas, tais como compiladores, ligadores (*linkers*), montadores, depuradores e ferramentas de avaliação temporal. Os resultados destas operações são apresentados graficamente para o usuário de maneira simples e intuitiva, como pode ser visto na ilustração da interface gráfica do PERF na Figura 2.

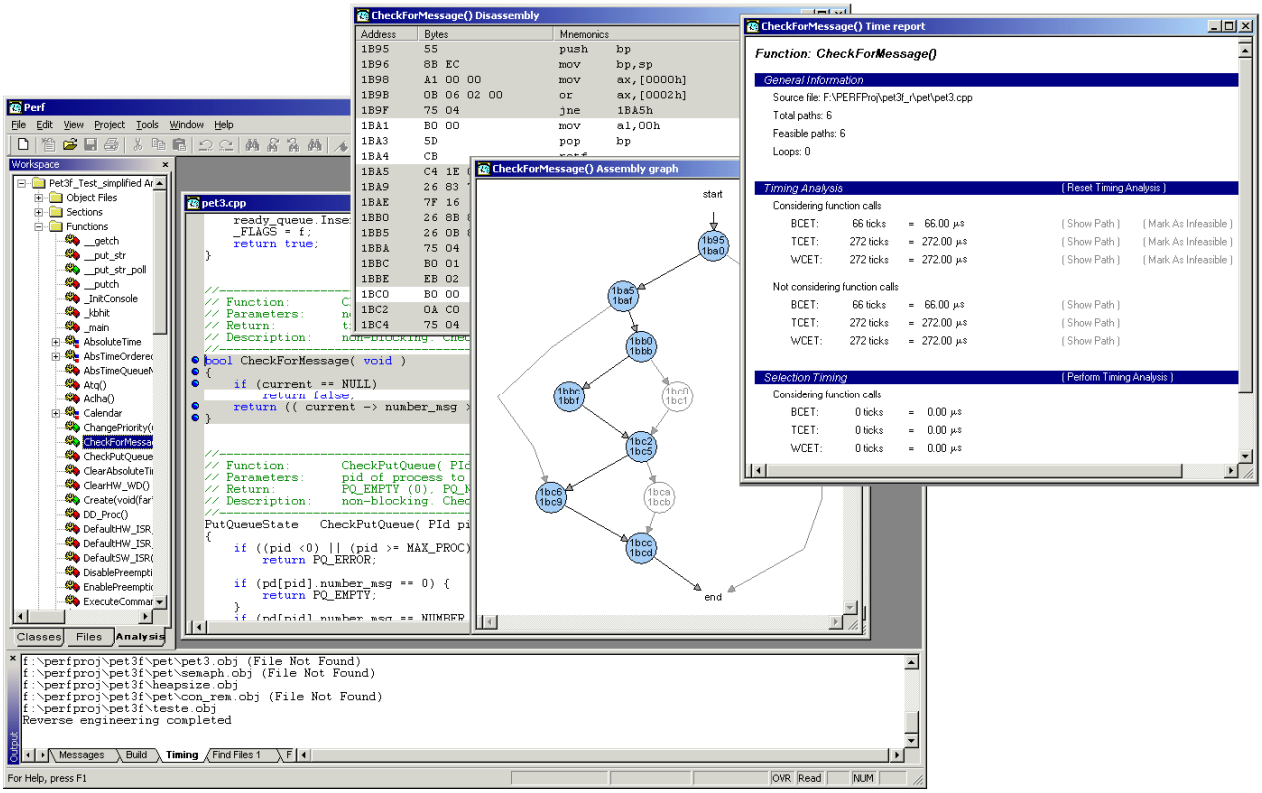


Figura 2: Interface gráfica do ambiente PERF**.**

Fonte: [Góes 01a]

Os conceitos e as idéias relacionados com o PERF estão em desenvolvimento desde 1992. Um histórico da evolução do PERF nas suas diferentes fases encontra-se na Tabela 1.

Tabela 1: Histórico do desenvolvimento do PERF.

Ano	Eventos relevantes
1992	Avaliação temporal da linguagem RTX-Parlog executando na arquitetura SPARC. A linguagem de programação lógica concorrente RTX-Parlog foi desenvolvida [Renaux 93] bem como um ambiente que inclui uma ferramenta de avaliação temporal.
1995	Medição temporal de *software* instrumentado utilizando analisador lógico [Renaux 99].
1996	Medição temporal das atividades de um núcleo de tempo real – PET [Renaux 96] utilizando o relógio de tempo real do próprio PET e armazenando um histórico de atividades na memória de trabalho do PET. Uma interface homem-máquina, implementada através de linha de comando, permite a visualização deste histórico [Renaux 99]
1998	TLM-OS (Time Line Monitor – Operating System): Monitoração temporal e funcional das atividades do PET, com apresentação de resultados na forma de linhas de tempo [Braga 98]
1998	TLM-MS (Time Line Monitor – Macro Segmentos): Monitoração temporal de trechos de código instrumentado [Braga 98]
1999	PERF – Definição da arquitetura do ambiente PERF para o desenvolvimento e avaliação temporal de STR. A concepção arquitetônica do PERF prevê a utilização de diversas ferramentas, tanto para atividades de desenvolvimento como para medição e estimação temporal. A arquitetura PERF também prevê a utilização de um banco de dados orientado a objetos como elemento de integração de ferramentas e estações de trabalho. [Renaux 99]
1999	TLM – Uma Ferramenta de Apoio ao Teste de Restrições Temporais em Sistemas Dedicados Operando em Tempo Real [Braga 99a][Braga 99b].
1999	Análise estática de código fonte visando estimação temporal [Kawamura 99]
2000	Análise estática de arquivos objeto e arquivos de biblioteca visando estimação temporal [Góes 00]
2001	O ambiente PERF possui a capacidade de análise estática de código do processador Intel 80C186EC, realizando estimativas dos tempos de execução com precisão melhor do que 5%. O PERF interage com o programador permitindo uma análise detalhada dos pontos críticos do código, e também gerencia projetos e o processo de edição e compilação dos mesmos [Góes 01a][Góes 01b][Renaux 02].
2001	Incluída a análise estática de código do processador Motorola (atual Freescale) MPC8xx, um RISC da família PowerPC. Também foi incluído o modelamento de hardware avançado [Linhares 01][Renaux 02].
2002	Introduzida a análise de escalonabilidade [Bregant 02].
2005	Adaptação do LTT (Linux Trace Toolkit) para a visualização da dinâmica de sistemas embarcados que não se baseiam em Linux [Bregant 05].
2006	Estudo sobre formas de visualização da dinâmica de sistemas embarcados implementados com programação concorrente [Navarro 06].
2006	Estudo sobre formas de escalonamento em ERTS [Luguesi 06].

Na Tabela 1 observa-se que a avaliação de um sistema no PERF por métodos estáticos foi desenvolvida a partir de dissertações de mestrado no CPGEI/UTFPR [Kawamura 99][Góes 01a][Linhares 01]. A estratégia dos métodos estáticos no PERF é encontrar os possíveis caminhos nos arquivos-objetos gerados pelo compilador, e então calcular o tempo de execução de cada caminho. Calculados os tempos de cada caminho, é possível determinar, para cada serviço ou função do sistema, o valor do BCET (*Best Case Execution Time*), do TCET (*Typical Case Execution Time*) e do WCET. O BCET é o valor de tempo mais otimista para a execução de uma função ou serviço, o TCET é uma estimativa do tempo médio de execução, e o WCET é o valor de tempo mais pessimista para se executar um trecho do programa. A validação do sistema é feita tomando por base o WCET, mas como este valor pode ser muito pessimista, aplicações práticas mostram que os valores do BCET e do TCET também têm importância. Como a abordagem baseia-se em arquivos-objetos, o PERF é capaz de calcular os tempos de execução de funções de biblioteca. Ainda como resultado das pesquisas acadêmicas que originaram o PERF, também é possível melhorar a precisão dos resultados dos tempos de execução estimados através da utilização de um modelo avançado de *hardware*. Este modelo avançado ajuda a descrever com maior precisão a plataforma alvo, incluindo nas estimativas informações sobre *pipeline*, memória *cache* de instruções e velocidade de acesso das memórias de trabalho [Linhares 01].

O método dinâmico utilizado pelo PERF, e que complementa o trabalho feito pelos métodos estáticos, é baseado em uma ferramenta chamada TLM (*Time Line Monitor*) [Braga 99a]. O TLM monitora programas escritos em C ou C++ que executem em sistemas embarcados baseados no microcontrolador Intel 80C186EC, com freqüência máxima de 25MHz. As informações de monitoração são coletadas do SUT por um monitor híbrido. Após o processamento das informações de monitoração, os resultados são apresentados ao usuário de quatro formas diferentes: linhas de tempo, grafo de segmentos, histogramas e código fonte. Outros detalhes sobre o TLM serão apresentados posteriormente na Seção 4.4.1. Após o TLM, houve também trabalhos relativos a métodos dinâmicos no sentido de experimentar e estudar formas de visualização do comportamento dinâmico de sistemas embarcados que são implementados com programação concorrente [Bregant 05] [Navarro 06].

2.5 RESUMO

Neste capítulo foram apresentados três tópicos em engenharia de *software* ligados ao desenvolvimento e à verificação de sistemas em tempo real: processo de desenvolvimento, etapa de teste e determinação do tempo de execução. O processo de desenvolvimento apresentado foi o utilizado no LIT. A seguir foram apresentados definições, termos e conceitos relacionados ao teste de *software*, bem como as especificidades relacionadas com o teste de *software* para sistemas em tempo real. Em seguida foram discutidos os métodos existentes para determinar o tempo de execução de um programa, que podem ser estáticos ou dinâmicos. Os métodos estáticos estimam o tempo através de cálculos analíticos, enquanto os métodos dinâmicos determinam o tempo através da monitoração do programa em execução. Com o objetivo de auxiliar o desenvolvimento de ERTS, o LIT vem trabalhando no projeto PERF, que é composto por um ambiente de desenvolvimento e um conjunto de métodos e ferramentas. O PERF induz o usuário a utilizar métodos estáticos e dinâmicos de maneira integrada e complementar. O Método Dyretiva é parte do PERF, ligado a verificação de ERTS por métodos dinâmicos.

No próximo capítulo será abordado em detalhes um assunto que foi postergado neste: a medição de tempos de execução. A medição é a forma como os métodos dinâmicos são capazes de determinar tempos de execução e, em razão disso, tem grande importância para o Dyretiva.

3 MEDIÇÃO DE TEMPOS DE EXECUÇÃO

A monitoração realiza a medição do tempo efetivamente gasto por um programa durante sua execução. A partir de dados capturados pela monitoração é possível obter informações sobre o sistema, tais como: taxa de utilização do processador, interações entre tarefas, o comportamento interno de cada tarefa, atividades de *hardware* e tempos de execução das tarefas.

A monitoração de um sistema normalmente produz um grande volume de dados. Para que se obtenham informações a partir dos dados coletados é necessário que estes sejam filtrados e analisados antes de serem apresentados ao usuário. Deste modo, o usuário recebe apenas as informações que são relevantes para verificar o correto funcionamento ou detectar um problema em uma IUT.

Na maioria dos sistemas reais a aplicação de métodos de monitoração para a verificação do sistema durante a fase de testes não garante que a IUT fique isenta de falhas. No entanto, a partir das informações extraídas dos dados de monitoração é possível garantir que a IUT se comporta de acordo com as especificações nas situações exercitadas pelos casos de teste utilizados. Entretanto se os casos de teste forem mal planejados ou pouco abrangentes é possível que falhas sejam encontradas futuramente no sistema, mesmo após extensivo período de testes.

3.1 CONCEITOS FUNDAMENTAIS DE MONITORAÇÃO

O conceito de monitoração exige a definição dos seguintes termos: ação, evento, sensor, instruções de instrumentação ou instrumentação, monitor e intrusão. Neste trabalho, estas definições foram extraídas principalmente do trabalho de Schroeder [Schroeder 95].

Uma **ação** é toda e qualquer atividade relacionada a um programa. São exemplos de ações a execução de procedimentos, funções, métodos ou serviços do sistema operacional. Toda ação possui uma duração associada a ela.

Um **evento** é um acontecimento de duração nula relacionado a um dado do programa ou a uma ação. Ele é caracterizado pelo instante de tempo em que o dado teve o seu valor alterado ou então pelo início ou término da execução de uma ação. Eventos relacionados a ações podem ser agrupados em três categorias: eventos de *hardware*, eventos em nível de tarefas e eventos em nível de aplicação.

Eventos de *hardware* são atividades tais como um pedido de interrupção, uma solicitação de uso de barramento ou uma falha de página. Eventos em nível de tarefas são aqueles que podem ser observados por uma entidade externa a tarefa, tais como o envio ou o recebimento de uma mensagem. Eventos em nível de aplicação descrevem o comportamento interno do programa de um usuário, dependendo basicamente do programador.

Um **sensor** é uma entidade que observa o comportamento de uma pequena parte da aplicação. Quando disparado, um sensor gera um evento. O sensor pode ser disparado tanto por uma mudança de estado na entidade que ele observa como por um pedido do sistema de monitoração. Quando disparado por uma mudança de estado na entidade observada, o sensor é dito marcador da entidade. A marcação é feita de maneira síncrona com a mudança de estado da entidade, ou seja, quando o valor muda, o sensor relata imediatamente o novo valor ao sistema de monitoração. Quando disparado por um pedido do sistema de monitoração, o sensor é dito de amostragem. Neste caso, o relato é feito assincronamente com relação à mudança de estado na entidade monitorada, sendo disparado quando a rotina de monitoração decide coletar os dados relativos àquela entidade e envia uma mensagem ao sensor apropriado, que então retorna o valor atual para o monitor.

Um sistema pode ser monitorado de duas formas distintas: por rastreamento (*tracing*) ou por contagem (*profiling*). O rastreamento obtém uma lista de todos os eventos de interesse ocorridos no sistema. A contagem apenas indica quantas vezes cada ação ocorreu. Para evitar que a monitoração insira instruções de instrumentação em todos os pontos de tomada de decisão no grafo de controle de fluxo do programa, Ball e Larus [Larus 94] propuseram algoritmos que procuram minimizar o número de instruções de instrumentação adicionadas ao código, sem prejuízo para as atividades de rastreamento ou contagem. Mais recentemente, estudos também procuram por formas mais eficientes de armazenar e manipular o grande volume de dados gerados pela monitoração. As soluções baseiam-se na reorganização dos eventos e na compressão dos dados coletados [Larus 99] [Gupta 01].

Os **monitores** são componentes de *software* ou de *hardware* que têm a função de capturar eventos, associar um rótulo temporal e possivelmente outros rótulos, e então armazenar a informação para análise posterior. Devido ao grande número de eventos gerados por um programa monitorado, o armazenamento normalmente é feito em um arquivo. O conjunto dos eventos capturados pelo monitor recebe o nome de arquivo de rastreamento (*trace*). Durante uma execução monitorada, os comandos ou rotinas de monitoração inseridos no programa funcionam como sensores que informam a ocorrência de eventos.

Estes comandos ou rotinas são chamados **instruções de instrumentação**, **instruções de monitoração** ou simplesmente **instrumentação**.

A instrumentação pode estar, além de inserida diretamente no programa monitorado, concentrada no sistema operacional ou então em uma única tarefa de monitoração, que executa concorrentemente às demais. A instrumentação inserida diretamente no programa monitorado é a mais simples e flexível, pois permite que o usuário colete apenas os dados de seu interesse. As outras duas formas não exigem que o programa do usuário seja alterado, pois capturam apenas as atividades do sistema operacional e as interações entre as tarefas.

A **intrusão** em um sistema monitorado é definida como o nível de interferência que a monitoração impõe sobre a aplicação. A intrusão pode ser vista como a utilização de recursos que poderiam ser utilizados pela aplicação para fins de monitoração, tais como tempo de processamento, espaço de armazenamento e canais de comunicação. Apesar de a intrusão ser indesejável, ela é uma forma viável de se obter informações sobre o comportamento dinâmico da aplicação.

3.2 CLASSIFICAÇÃO DOS MONITORES

A monitoração de programas é uma área que vem experimentando avanços. Na década de 1960, monitorar um programa era sinônimo de executá-lo sob a supervisão de um depurador de código (*debugger*), conceito que precisou evoluir para acomodar outras formas de acompanhar a execução de um programa. Uma classificação de monitores feita por Schroeder [Schroeder 95] dividiu-os em sete categorias, conforme o seu objetivo: confiabilidade, melhoria de desempenho, correção, segurança, controle, teste e depuração e avaliação de desempenho. As características de cada categoria são apresentadas a seguir.

- **Confiabilidade**: é a monitoração feita com o objetivo de verificar que o sistema se mantenha dentro das especificações.
- **Melhoria de desempenho**: é a monitoração que visa à configuração dinâmica do sistema, o aprimoramento da dinâmica do programa e a navegação em tempo de execução.
- **Correção**: é a monitoração de uma aplicação para certificar-se de sua consistência com uma especificação formal. Pode ser utilizada para detectar erros em tempo de execução ou apenas como técnica de verificação.

- **Segurança**: é a monitoração que se preocupa em detectar violações de segurança, tais como utilização por um usuário ilegal ou tentativa de acesso a arquivo restrito.
- **Controle**: são os casos em que o sistema de monitoração faz parte da plataforma alvo, ou seja, é um componente necessário para prover funcionalidade computacional.
- **Teste e depuração**: são monitores que extraem dados da aplicação para fins de validação lógica, ou seja, tem objetivos semelhantes aos depuradores de código.
- **Avaliação de desempenho**: é a monitoração voltada a extrair dados do SUT para uso na avaliação do desempenho do sistema.

É importante notar que um sistema de monitoração pode contemplar mais de uma das categorias apresentadas acima.

3.3 MÉTODOS DE MONITORAÇÃO

Os métodos de monitoração podem ser classificados em três tipos básicos: métodos baseados em *software*, métodos baseados em *hardware* e métodos híbridos [Stunkel 91][Schroeder 95][Mahrenholz 01]. Monitores baseados em *software* são basicamente extensões do sistema existente, escritas com a finalidade de capturar e registrar a ocorrência de eventos do programa. Ao contrário, os monitores baseados em *hardware* observam os sinais elétricos dos barramentos do sistema para rastrear as atividades que o *software* está executando, implicando necessidade de acesso físico ao SUT. Os monitores híbridos são soluções de compromisso que procuram utilizar as melhores características dos dois tipos anteriores: executam o mínimo indispensável no sistema e providenciam uma forma de realizar as demais tarefas de monitoração através de um *hardware* externo. A Figura 3 apresenta um resumo da classificação dos monitores.

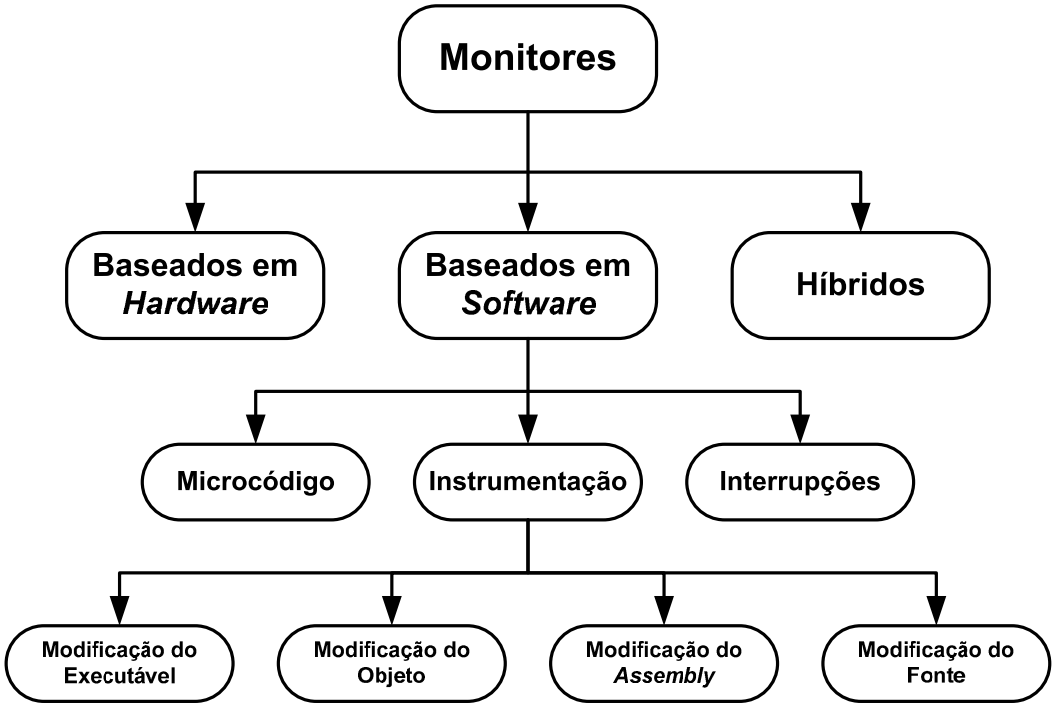


Figura 3: Classificação dos monitores.

Nas seções a seguir são apresentados detalhes sobre cada uma das três classes de monitores.

3.3.1 MONITORAÇÃO BASEADA EM HARDWARE

A monitoração baseada em *hardware* é um método voltado a observar os sinais elétricos dos barramentos de endereços, dados e controle do SUT para verificar as referências externas que são feitas pelo processador. Os dados coletados por esta observação precisam ser correlacionados de alguma forma com o código do programa em execução, que é a parte do sistema com a qual o desenvolvedor se identifica. A implementação de um monitor baseado em *hardware* é considerada complexa porque não existe maneira trivial de realizar esta correlação. Além disso, o uso de MMU (*Memory Management Unit*) tem se tornado comum em sistemas embarcados, sendo sua presença essencial para muitos sistemas operacionais. Quando a MMU é ativada, os endereços dos programas tornam-se virtuais, ao contrário dos endereços que aparecem nos barramentos, que são reais ou físicos. Com isso o grau de dificuldade de correlacionar as referências externas feitas durante a execução de um programa com o seu código fonte aumenta ainda mais.

Uma vantagem da monitoração baseada em *hardware* é o fato da coleta de dados ser completamente não intrusiva do ponto de vista do *software*. Esta característica evita que a IUT precise ser instrumentada, utilizando tempo de processamento para a execução das instruções de instrumentação. Uma outra vantagem intrínseca deste método é a capacidade de acompanhar eventos de *hardware* que não são percebidos pelo *software*, tais como solicitações de interrupções e uso do barramento para ciclos de DMA, que interferem no desempenho do sistema.

Por outro lado existem várias desvantagens na monitoração baseada em *hardware*. Uma delas deve-se ao fato de existir grande dependência entre o monitor e a arquitetura do processador utilizado. Arbitramento de barramento, sinais de controle e arquitetura da memória normalmente são únicos para uma arquitetura ou até mesmo modelo de processador, fazendo com que a migração para um novo modelo ou arquitetura exija grandes investimentos em um novo monitor. Este motivo dá aos monitores baseados em *hardware* características de baixa flexibilidade e alto custo. Além disso, existem ainda algumas características que vêm sendo agregadas aos processadores modernos e que têm tornado este tipo de monitor praticamente inviável. Entre estas podemos citar a predição de desvios (*branch prediction*) e as memórias integradas. As memórias integradas – sejam elas RAM estática, *cache* ou *flash* – têm o objetivo de melhorar a velocidade de acesso do processador aos dados que o programa necessita e também de diminuir os níveis de emissão de energia eletromagnética gerados pelo chaveamento dos circuitos eletrônicos. Para que estes objetivos sejam atingidos é necessário que os ciclos de acesso a estas memórias não apareçam nos barramentos externos do processador, impedindo sua amostragem por um agente de *hardware* externo. Além disso, a predição de desvios pode decidir pela busca e início da execução de partes do código que mais tarde serão descartadas em virtude da predição ter se equivocado. Nestes casos, o acompanhamento da execução de um programa via observação dos sinais do barramento agrega também estes problemas, além dos outros já citados anteriormente.

A monitoração de um sistema por *hardware* pode ser feita de duas formas: construindo um equipamento dedicado para realizar a coleta de informações em uma determinada plataforma ou arquitetura, ou utilizando um equipamento de medição disponível comercialmente. Um exemplo de equipamento comercial utilizado para coletar informações por *hardware* são os analisadores lógicos que, quando conectados aos sinais digitais de um sistema eletrônico, são capazes de amostrá-los. O objetivo primordial de um analisador lógico não é realizar monitoração por *hardware*, mas sim observar o comportamento dos

sinais digitais do circuito. No entanto, como o suporte de *hardware* necessário para fazer a monitoração está presente no analisador, o equipamento pode também ser utilizado para esta finalidade. Para preencher a lacuna que existe entre os dados coletados pelo analisador e o código fonte do programa, os fabricantes destes equipamentos disponibilizam módulos de *software* que, baseados no modelo de processador, realizam a correlação entre os sinais do barramento e o código fonte do usuário. Estes módulos são dependentes do modelo do processador porque existe uma estreita ligação entre a linguagem de máquina do processador e os sinais de seu barramento. Dependendo das características construtivas de um determinado processador, é possível que tais módulos de *software* não possam ser construídos para ele. Podem ser citados como exemplos de analisadores lógicos capazes de realizar estas tarefas o TLA7016 [Tektronix 05] e o 16902A [Agilent 05].

Em resumo, a monitoração baseada em *hardware* apresenta vantagens únicas como intrusão em nível de *software* nula e observação de eventos de *hardware* que não são detectáveis por monitoração baseada em *software*. Por outro lado existem alguns outros fatores que desestimulam o uso deste tipo de monitoração, dentre os quais podemos citar: a alta complexidade, a grande dependência da arquitetura do processador, que implica em baixa flexibilidade, e alto custo. Além disso, a captura dos eventos com o uso de analisadores lógicos é limitada pela profundidade da memória de armazenamento do equipamento, o que pode se refletir em número de eventos insuficientes para analisar um problema de *software*.

A partir de 1998, companhias como Motorola Semicondutores (atual Freescale) e Agilent Technologies perceberam a necessidade de realizar a monitoração mais eficiente de programas que executam em microprocessadores de alto desempenho e com alto nível de integração. Neste ano, algumas empresas formaram um consórcio para a definição de uma "interface padronizada para depuração de sistemas embarcados". Posteriormente este consórcio recebeu a adesão de outros membros, tanto fabricantes de semicondutores como fornecedores de equipamentos e soluções, e o padrão foi então adotado pelo Instituto dos Engenheiros Eletricistas e Eletrônicos (IEEE), sendo batizado de Nexus 5001 [Dees 05]. A versão mais recente do padrão Nexus 5001 foi publicada no ano de 2003 [Nexus 03].

A proposta do Nexus 5001 é incorporar ao silício do processador o suporte para as tarefas de depuração e de monitoração, de forma consistente e padronizada. O usuário poderia se beneficiar do Nexus por ter uma visão uniforme das facilidades oferecidas por um processador, independentemente de seu modelo ou fabricante [Turley 04]. As empresas que fornecem ferramentas de depuração e de monitoração poderiam se beneficiar pela redução

nos custos de desenvolvimento das ferramentas, bem como da possível expansão dos negócios através do suporte a outros modelos e famílias de processadores. Com o Nexus 5001 os problemas típicos do monitoramento baseado em *hardware*, tais como a dificuldade de manipulação da MMU, da predição de desvios e da memória *cache*, bem como a falta de portabilidade e o alto custo, poderiam ser resolvidos. Apesar das vantagens oferecidas a usuários, fabricantes de semicondutores e fornecedores de ferramentas de desenvolvimento, a adoção ampla do Nexus 5001, ou de um padrão semelhante para efeitos de monitoração baseada em *hardware*, ainda parece ser uma realidade distante.

3.3.2 MONITORAÇÃO BASEADA EM SOFTWARE

A monitoração baseada em *software* é um método que modifica os programas que executam no SUT com a finalidade registrar eventos relevantes, permitindo assim o acompanhamento da execução do programa. Uma virtude importante dos monitores baseados em *software* é que sua implantação é mais barata e mais simples do que a dos monitores baseados em *hardware*, pois não exigem acesso físico ao SUT, além de serem relativamente independentes da arquitetura do processador utilizado. Entretanto, este tipo de monitoração apresenta a desvantagem intrínseca de ser muito intrusiva, e sua eficiência depende do método utilizado para a coleta dos dados e da freqüência de execução da monitoração. Nos casos em que a monitoração altera significativamente o desempenho do sistema, os dados capturados podem ser de pouca ou até mesmo de nenhuma utilidade para a avaliação temporal ou de desempenho do sistema.

Uma característica da monitoração baseada em *software* é que os dados capturados pela monitoração precisam ser tratados e armazenados na própria plataforma alvo, o que implica reserva de recursos de processamento e armazenamento. No caso específico dos sistemas embarcados, cujos recursos normalmente são escassos, é comum que a plataforma alvo não tenha a capacidade de processamento e as ferramentas necessárias para tratar os dados coletados. Nestes casos, é necessário ainda que exista um canal de comunicação entre o sistema embarcado e um computador externo, para onde as informações de monitoração serão enviadas para análise.

A monitoração baseada em *software* pode ser implementada de diferentes maneiras, dentre as quais serão apresentadas a seguir: monitoração por interrupção, monitoração baseada em alteração do microcódigo e monitoração por instrumentação.

A **monitoração por interrupção** é baseada na capacidade dos processadores de desviar o fluxo do programa para uma rotina do sistema operacional ou do usuário após a execução de cada instrução. Este mecanismo é conhecido como "interrupção passo-a-passo". Também é possível, de forma similar, desviar o fluxo de execução de um programa para uma rotina do sistema operacional ou do usuário quando se executa uma instrução especial do processador (*trap*) ou uma interrupção de *software*. É utilizando este mecanismo que os depuradores de código convencionais (*debuggers*) implementam a inserção de pontos de parada (*breakpoints*) e de pontos de coleta de dados (*tracepoints* e *watchpoints*) no código do usuário. No entanto, a monitoração por interrupção torna a execução monitorada do programa muitas vezes mais lenta do que a sua execução original, inviabilizando o uso desta técnica para fins de avaliação temporal e de desempenho.

A **monitoração baseada em alteração do microcódigo** do processador é uma outra forma de realizar a captura das informações necessárias para a monitoração. Utilizando esta técnica é possível analisar tanto a aplicação como o sistema operacional, contando ainda com uma intrusão muito menor do que a causada por interrupções de *software*. O problema neste caso é que o microcódigo dos processadores comerciais nem sempre está disponível e, mesmo nos casos onde é possível obtê-lo, sua alteração é uma tarefa demorada, delicada, trabalhosa e que exige muitos conhecimentos específicos relativos ao projeto de microprocessadores.

Das três técnicas de monitoração baseadas em *software* que foram citadas, a **monitoração por instrumentação** é a mais conhecida e utilizada. Sua apresentação mais usual é o envio de uma mensagem de texto para um terminal ou para um arquivo, contendo informações sobre a ocorrência de um determinado evento. O conjunto de todas as mensagens geradas pelo programa é o registro dos eventos monitorados. Muitos sistemas simples podem ser monitorados desta forma, ficando a cargo do programador a interpretação do registro para a identificação de falhas.

Existem também outras formas de realizar a monitoração por instrumentação. Uma delas consiste em dividir o código em blocos básicos, que são conjuntos de instruções que sempre executam seqüencialmente. Dentro de um bloco básico não devem existir desvios ou chamadas de sub-rotinas. No início e/ou o final de cada bloco básico são inseridas instruções de instrumentação de forma que seja possível registrar a ocorrência do bloco e determinar, por exemplo, o seu tempo de execução.

Em todas as formas de monitoração por instrumentação é necessário modificar o código do programa para que as instruções de instrumentação possam ser inseridas. A modificação do código pode ser feita em quatro níveis distintos: modificação do programa executável, modificação dos arquivos objeto, modificação do código de máquina ou modificação dos arquivos fonte.

A monitoração por instrumentação baseada na modificação do programa executável consiste em desmontar o arquivo binário, localizar os blocos básicos que se deseja observar, incluir as instruções de instrumentação e remontar o arquivo binário. A principal vantagem desta abordagem é que ela torna possível a observação de um programa mesmo quando o código fonte ou o código objeto não estão disponíveis. Entretanto, a modificação do código executável é uma tarefa complexa porque exige conhecimento detalhado tanto do formato do arquivo executável como da linguagem de máquina do processador. Na prática esta abordagem só é viável para processadores RISC muito simples, e ainda nos casos onde a licença de utilização do programa que se deseja analisar não proíbe tal prática. Para processadores como o popular x86 da Intel, que possui endereços ou deslocamentos relativos de tamanho variável e de forma entrelaçada com as instruções, a modificação de um programa executável utilizando esta abordagem é muito complexa.

A monitoração por instrumentação baseada na modificação de arquivos objeto foi uma técnica proposta por Cargille e Miller [Cargille 92] como uma solução para a monitoração de chamadas ao sistema operacional e de funções de biblioteca, sem a necessidade de modificar o código fonte do usuário, o código fonte do sistema operacional, ou o código fonte da biblioteca. Nesta técnica as tabelas de símbolos dos arquivos objeto são modificadas de modo a renomear a rotina do sistema operacional ou da biblioteca que se deseja monitorar. Após renomear a rotina original, uma nova rotina é criada com o mesmo nome da rotina original. Esta nova rotina tem a função de realizar as tarefas de monitoração que se fizerem necessárias, chamando em seguida a rotina original que foi renomeada. No retorno da rotina original ainda é possível realizar eventuais tarefas finais de monitoração antes de retornar para a rotina que invocou o serviço do sistema operacional ou a função da biblioteca. As tarefas de monitoração executadas por esta técnica dependem de implementação específica, mas podem ser desde um simples aviso de que a rotina foi chamada até o registro dos parâmetros utilizados em cada chamada, ou mesmo o cálculo do tempo de execução.

A aplicação desta técnica pode ser exemplificada através de uma chamada à função **open** do sistema operacional UNIX. Primeiramente a função **open** original precisa ser localizada na biblioteca do sistema e renomeada para **_open**. A partir daí uma outra função **open** precisa ser criada, contendo as informações de instrumentação que se deseja incluir e a chamada à função **_open** que foi renomeada. Deste modo a instrumentação é incluída sem modificar nem o código do usuário e nem o código da função **open** da biblioteca. A Figura 4 ilustra esta técnica utilizando o exemplo descrito.

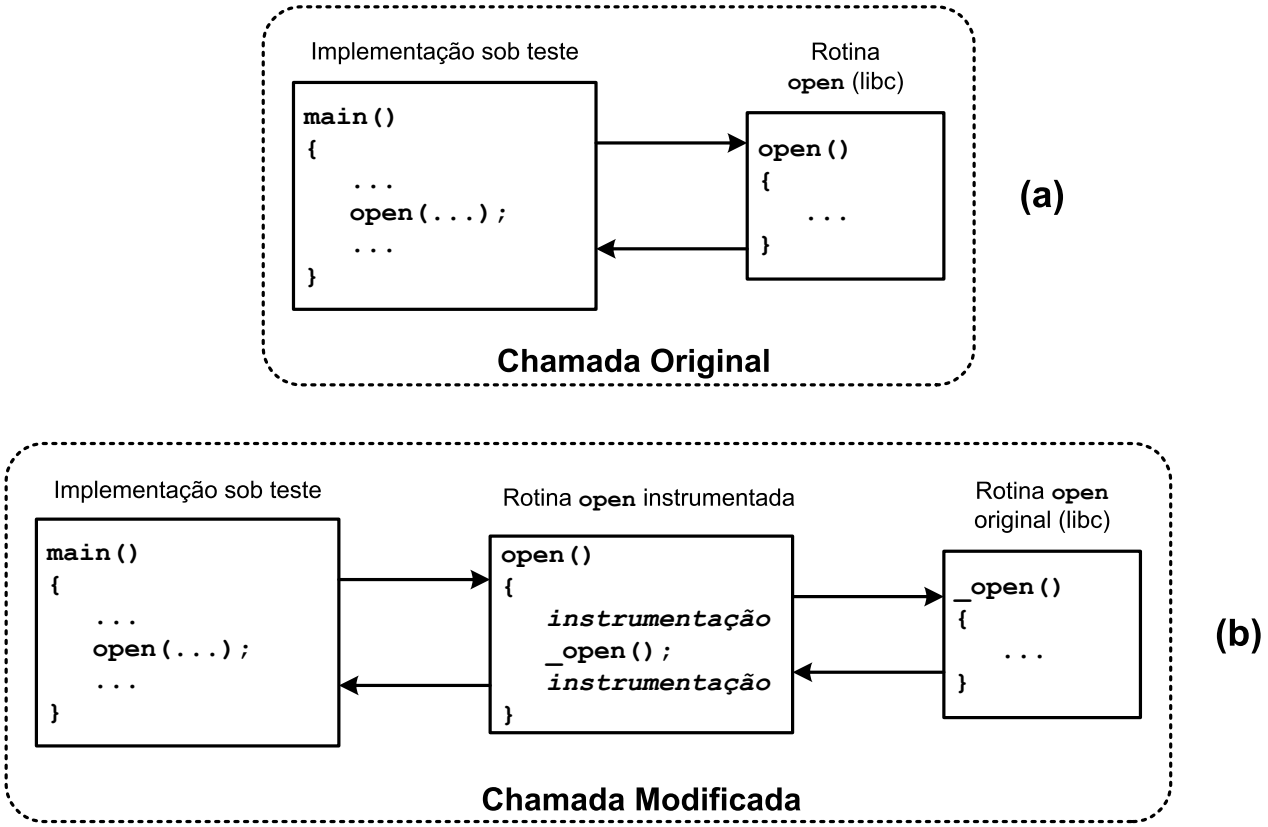


Figura 4: Exemplo de aplicação da técnica de modificação de arquivos objeto.
Fonte: [Cargille 92]

A monitoração por instrumentação baseada na modificação do código de máquina (*assembly*) é uma opção à complexidade envolvida na modificação do código executável, e que é capaz de realizar mais do que a monitoração apenas de chamadas de função proposta na modificação dos arquivos objeto. A localização de blocos básicos dentro do código de máquina de um programa é uma tarefa viável e que torna possível localizar inclusive tomadas de decisão que não eram visíveis no código fonte, e que foram inseridas no código

de máquina pelo compilador para implementar uma determinada estrutura de alto nível do programa. Esta técnica também tem a vantagem de tornar a instrumentação imune às otimizações feitas pelos compiladores sobre as linguagens de alto nível.

Uma desvantagem da modificação do código de máquina é a dependência da arquitetura utilizada, uma vez que a localização dos blocos básicos e a instrumentação precisam ser incluídas na linguagem de montagem (*assembly*). Outra desvantagem é que, quando são utilizados programas de terceiros para a construção de uma solução, torna-se difícil – se não impossível – obter o código de máquina desta parte da solução para que se possa inserir a instrumentação.

A monitoração por instrumentação baseada na modificação do código fonte é a mais popular das formas de monitoração baseadas instrumentação. Esta técnica pode ser aplicada de forma manual pelo próprio programador ou, então, de forma automática através do uso de um instrumentador de código. Os instrumentadores de código são compiladores simplificados que identificam os blocos básicos no código fonte e inserem as instruções de instrumentação apropriadas. A saída do instrumentador é um novo arquivo fonte, agora instrumentado, que será então compilado e ligado para gerar o programa executável. Uma das desvantagens da modificação do código fonte é que não é possível realizar a instrumentação quando os arquivos fonte não estão disponíveis, o que inclui as bibliotecas distribuídas na forma de arquivos objeto.

Como a instrumentação automática gera um grande volume de dados durante o rastreamento, técnicas de filtragem precisam ser aplicadas para extrair apenas as informações de interesse para a análise de um determinado problema. Além disso, existem ainda estudos no sentido de armazenar as informações geradas pela instrumentação de forma compactada, economizando espaço e facilitando o acesso a uma instância qualquer de uma determinada ação [Larus 99] [Gupta 01].

3.3.3 MONITORAÇÃO HÍBRIDA

Os monitores híbridos utilizam componentes de *hardware* e de *software* para realizar a monitoração, buscando aproveitar as vantagens tanto de monitores baseados em *hardware* como de monitores baseados em *software*. Os monitores híbridos são classificados em internos e externos, conforme o responsável pelo armazenamento e processamento dos dados coletados: o próprio SUT no primeiro caso, e um *hardware* externo no segundo.

O componente de *software* do monitor híbrido é executado junto com a aplicação do usuário e o sistema operacional no SUT, da mesma forma que seria feito por um monitor baseado em *software*. No entanto, para preservar apenas as melhores características dos monitores baseados em *software*, a única responsabilidade deste componente é sinalizar eventos para que sejam capturados e temporizados pelo componente de *hardware* do monitor. Com isto a intrusão no SUT pode ser minimizada, dependendo basicamente da quantidade de eventos inserida e da eficiência de sua sinalização para o componente de *hardware* do monitor. O componente de *software* do monitor híbrido também preserva outra importante característica dos monitores baseados em *software*: a independência da arquitetura do processador.

A responsabilidade do componente de *hardware* do monitor híbrido é capturar e marcar o tempo dos eventos com alta velocidade, precisão e sem interferir no *software* do SUT. Deste modo, o SUT é desobrigado das tarefas de monitoração que mais consomem tempo de processamento, memória e canais de comunicação. Com isto os recursos SUT podem ser dimensionados apenas para as suas próprias necessidades, e não para as da monitoração, apresentando as vantagens de: diminuição do tempo no desenvolvimento, minimização de recursos adicionais que seriam necessários apenas para efetuar monitoração, e redução no custo final do *hardware*. Apesar disso, é importante frisar que a monitoração híbrida externa precisa de conexão física com o SUT, de forma que o custo de *hardware* no SUT associado à monitoração caso não é nulo e depende, basicamente, da forma prevista para sinalização de eventos entre o componente de *software* e o componente de *hardware* do monitor externo.

Em resumo, a monitoração híbrida apresenta as seguintes vantagens: baixa intrusão, boa precisão, flexibilidade, portabilidade e mínima utilização de recursos de *hardware* e de *software* do SUT.

3.4 RESUMO

Neste capítulo foram apresentados os conceitos de monitoração da execução de programas, que são a base para a aplicação de métodos dinâmicos de verificação em sistemas embarcados com restrições temporais. Os monitores foram classificados conforme o seu objetivo: confiabilidade, melhoria de desempenho, correção, segurança, controle, teste e depuração e avaliação de desempenho. Os três métodos de monitoração existentes foram apresentados: monitoração baseada em *hardware*, monitoração baseada em *software* e monitoração híbrida. Para cada método de monitoração foram discutidas as diferentes formas de aplicação, bem como suas vantagens e desvantagens.

No próximo capítulo será apresentado o domínio do problema relacionado à verificação das restrições temporais em sistemas embarcados que operam em tempo real. Este próximo capítulo será fundamentado nos conceitos já apresentados tanto neste capítulo como no anterior, estabelecendo as bases, a abrangência e a ambiente de trabalho a ser utilizado no método de verificação proposto.

4 VERIFICAÇÃO TEMPORAL DE SISTEMAS EMBARCADOS

As atividades de teste e de verificação são fundamentais na garantia da qualidade do *software* embarcado que opera em tempo real, e precisam considerar tanto os aspectos funcionais quanto os temporais. No entanto, sistemas embarcados possuem recursos limitados (Capítulo 2), o que os torna dependentes da plataforma de *hardware* em que executam. Outra característica dos ERTS é a implementação com programação concorrente (Seção 2.2.2), o que os torna dependentes do ambiente de execução.

Este capítulo aborda a verificação temporal de ERTS, iniciando por uma descrição do ambiente de trabalho, que envolve as características das plataformas de *hardware* embarcadas, o ambiente de desenvolvimento e o ambiente de execução de *software*. O ambiente de trabalho descreve características intrínsecas dos sistemas embarcados, que serão utilizadas logo em seguida como limitador de escopo para a pesquisa dos trabalhos relacionados encontrados na literatura.

4.1 PLATAFORMA DE HARDWARE

As plataformas de *hardware* consideradas neste trabalho são aquelas equipadas com processadores dedicados de 16 ou de 32 bits. A classificação de um processador utilizando este critério se dá pela largura dos registradores internos e das entradas da unidade lógica e aritmética (ALU) [Henessy 96]. Para atender as limitações dos sistemas embarcados, estes processadores normalmente possuem periféricos integrados, podendo também ser equipados com memórias. Os sistemas desenvolvidos com eles possuem uma ou mais portas de comunicação, que proporcionam interações com outros sistemas, além de sensores e atuadores para interação com o ambiente.

Um exemplo de *hardware* embarcado que se encaixa na descrição acima é o módulo e850Lite [eSysTech 03], que é utilizado nos experimentos práticos deste trabalho. Um diagrama em blocos desta plataforma de *hardware* é mostrado na Figura 5. O módulo é composto por um microprocessador Freescale MPC850, que é uma implementação de 32-bits da arquitetura Power [Power 2006], operando a uma freqüência de 48MHz. O processador possui internamente 2KB de memória cache de instruções, 1KB de memória cache de dados, controlador para até 8 bancos de memória, além de diversos controladores de comunicação. O módulo e850Lite conta com 32MB de memória SDR-SDRAM, 8MB de

memória flash, duas portas seriais padrão RS-232, uma porta USB, uma porta Ethernet e uma saída paralela de 16-bits.

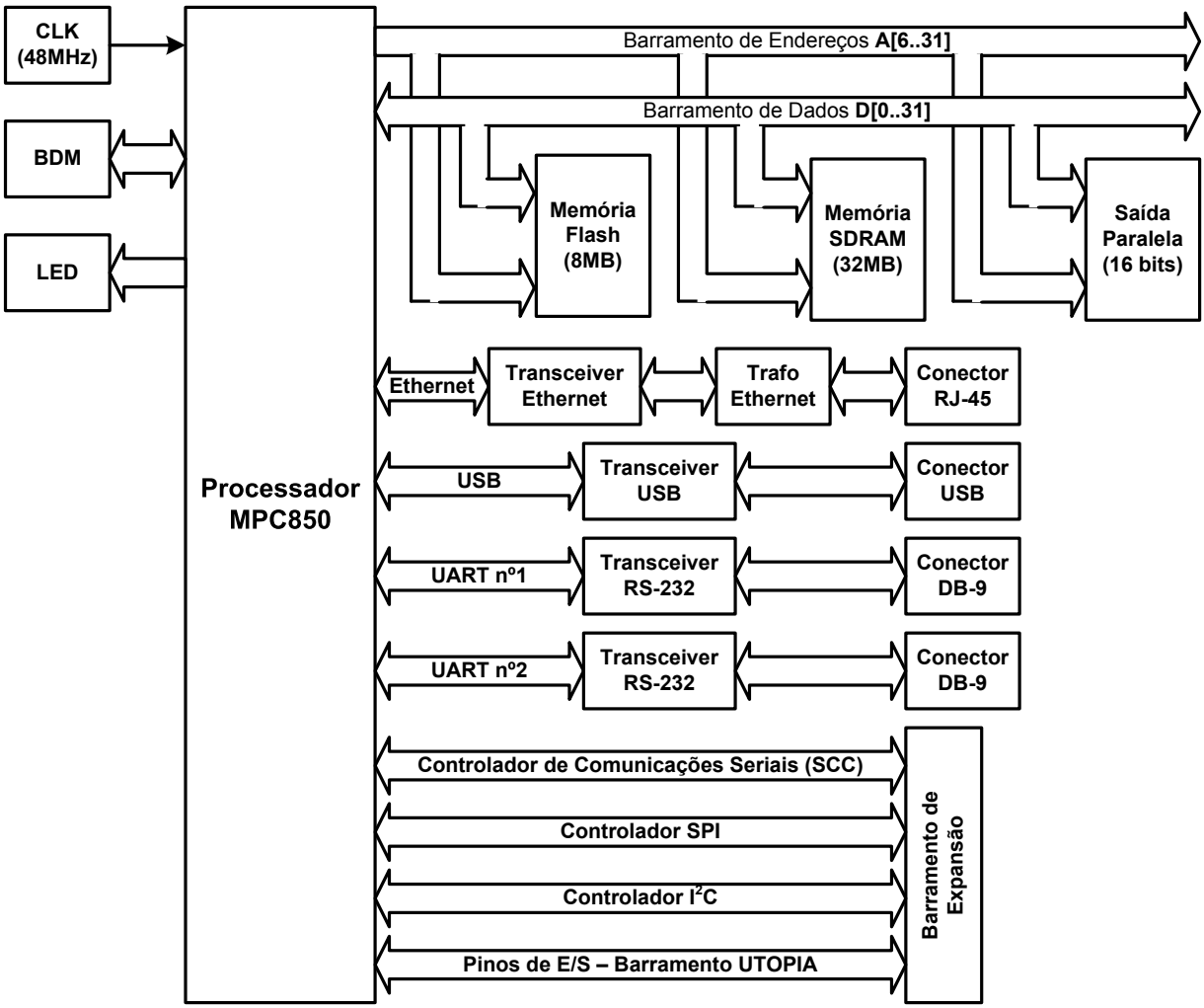


Figura 5: Diagrama em blocos do módulo e850Lite.

Fonte: [eSysTech 03]

4.2 AMBIENTE DE PROGRAMAÇÃO

Software embarcado freqüentemente é escrito em linguagem de programação C ou C++. Isto acontece porque C é uma linguagem de amplo espectro, ou seja, que possui tanto recursos de linguagens de alto nível, como estruturas de dados e bibliotecas, quanto recursos de baixo nível, como a possibilidade de acessar diretamente registradores de *hardware* dos periféricos. Entretanto, C é uma linguagem procedimental, o que leva muitos programadores a optarem por C++, que é uma linguagem de mais amplo espectro, incorporando conceitos

de orientação a objetos. O que torna o C++ viável para aplicações embarcadas é o fato de herdar toda a sintaxe da linguagem C, permitindo a realização de operações de baixo nível quando necessário. Para efeitos deste trabalho, considerar-se-á C como a linguagem de programação adotada, pois atende às necessidades dos sistemas embarcados, e possui a sintaxe mais simples dentre as duas.

Para gerar e depurar a aplicação são utilizados ambientes integrados de desenvolvimento (IDE), que permitem gerenciar e editar os arquivos fontes, gerar o programa executável e realizar atividades de depuração com o objetivo de encontrar erros de lógica. Um IDE pode se destinar ao desenvolvimento nativo ou ao desenvolvimento cruzado. No desenvolvimento nativo, tanto o IDE quanto o programa gerado por ele executam na mesma plataforma de *hardware*. Já no desenvolvimento cruzado, o IDE gera um programa para executar em uma plataforma de *hardware* diferente daquela utilizada por ele, que é a situação usual em sistemas embarcados.

Os programas desenvolvidos para o módulo e850Lite neste trabalho foram gerados e depurados com o MULTI 2000 IDE [Green 07], desenvolvido pela empresa Green Hills Software. O MULTI gera código para mais de uma dezena de arquiteturas de processadores, incluindo a arquitetura Power do processador MPC850.

Nos processadores modernos, a transferência do código binário gerado pelo IDE para a plataforma embarcada é feita através de módulos de *hardware* que existem nos microprocessadores, chamados de ICE (*In-Circuit Emulator*), que são acessíveis por interfaces seriais JTAG (*Joint Test Action Group*) ou OCD (*On-Chip Debug*). Os usuários comandam o ICE a partir do IDE utilizando dispositivos externos que convertem os dados vindos por interfaces padrão dos computadores, tais como portas seriais RS-232, portas paralelas IEEE-1284, USB ou Ethernet, em comunicação serial JTAG ou OCD. Utilizando estes recursos, o IDE tem a capacidade de gravar o código binário na memória não-volátil do sistema embarcado, ou carregá-lo diretamente na memória volátil para depuração. Neste último caso, o ICE e o depurador do IDE criam para o usuário um ambiente similar àquele utilizado por programas nativos. Para carregar e depurar os programas feitos com o MULTI para o módulo e850Lite foi utilizado um dispositivo externo chamado Wiggler [Macraigor 07], que converte dados vindos pela interface paralela IEEE-1284 de um computador pessoal em comunicação serial OCD para o processador MPC850.

4.3 AMBIENTE DE EXECUÇÃO

A Seção 2.2.2 abordou a questão de que os ERTS normalmente são implementados com programação concorrente, com suporte de um RTOS. Entretanto, para que se possa testar e verificar temporalmente um sistema que utiliza um RTOS é necessário ter visibilidade das atividades internas do núcleo, tais como chaveamentos de contexto, preempções e interrupções, pois são elas que determinam a seqüência e o tempo de execução das tarefas.

Para atingir o nível de conhecimento da estrutura interna do sistema operacional que permitisse uma observação adequada do seu comportamento, um núcleo de tempo real foi desenvolvido especialmente para os experimentos desta pesquisa. Este núcleo é o PET#, que é uma evolução do núcleo PET desenvolvido anteriormente no LIT [Renaux 96]. Outras informações sobre o PET# podem ser obtidas no Anexo I. O núcleo PET e seus derivados estão registrados no INPI (Instituto Nacional da Propriedade Industrial) sob o número de depósito 00074942 (2005).

Na família de núcleos de tempo real PET a comunicação e a sincronização entre as tarefas são feitas com trocas de mensagens. Segundo Burns, existem três métodos de trocas de mensagens entre tarefas [Burns 97], que podem ser descritos em função do comportamento do remetente. O primeiro modelo é o envio assíncrono, onde o remetente continua a executar após o envio, independente de a mensagem ter sido entregue ao destinatário ou não. O segundo modelo é o envio síncrono, onde o remetente é bloqueado no momento do envio, permanecendo neste estado até que o receptor receba a mensagem. O terceiro modelo é conhecido como chamada remota e requer que o destinatário, além de receber a mensagem, também responda ao remetente para que este seja desbloqueado. Os núcleos da família PET suportam dois deles: o envio assíncrono e a chamada remota. Utilizando as informações de monitoração do núcleo e as propriedades dos métodos de comunicação e sincronização utilizados pelo PET é possível reconstruir as linhas de execução de cada uma das tarefas do sistema, obtendo as informações necessárias para analisar o comportamento temporal do *software* embarcado.

A diferença mais relevante entre o PET e o PET# é o escalonador de tarefas preemptivo. Esta modificação foi necessária porque em ERTS uma forma comum de fazer a análise de escalonamento do sistema é através da abordagem da taxa monotônica, cujos resultados partem da premissa de que o escalonador é preemptivo [Briand 99].

Apesar de existirem outros RTOS que poderiam ter sido utilizados, a escolha de evoluir um núcleo de tempo real desenvolvido no LIT apresenta pelo menos três vantagens significativas. Primeiro, as características conhecidas, estudadas e comprovadas do PET em trabalhos anteriores continuam válidas. Segundo, as características atuais e as direções futuras do RTOS ficam sob o controle do grupo de pesquisa. Terceiro, o acesso ao código fonte para estudo e uso em laboratórios acadêmicos e salas de aula é garantido em longo prazo.

4.4 TRABALHOS RELACIONADOS

Durante a pesquisa bibliográfica, pesquisas acadêmicas e produtos com objetivos e características semelhantes aos do Dyretiva foram encontrados, dentre os quais cinco foram selecionadas para estudo mais detalhado: o TLM, o VDS, o CodeTEST, o WindView, e a Reprise utilizando Máquinas de Tempo.

4.4.1 TLM

O TLM (*Time Line Monitor*) é a ferramenta de análise dinâmica utilizada atualmente pelo ambiente PERF. Ele foi desenvolvido como dissertação de mestrado no CPGEI [Braga 99a], e é capaz de auxiliar no teste funcional, estrutural e temporal de um sistema embarcado. O TLM é composto por três ferramentas: um instrumentador de código fonte, um monitor híbrido, e um programa de rastreamento e apresentação de resultados. Um diagrama de contexto do TLM é mostrado na Figura 6.

O instrumentador do TLM é uma ferramenta em linha de comando que transforma o código fonte do usuário. Utilizando três tipos de comandos, que devem ser inseridos no código na forma de comentários de formato pré-determinado, o instrumentador é capaz de gerar instrumentação em três níveis: nenhuma instrumentação, instrumentação de funções ou instrumentação total, ou seja, de todas as funções, laços e tomadas de decisão. Baseado nestes comandos, o instrumentador transforma o código do usuário em um novo código fonte, agora contendo as instruções de instrumentação nos lugares solicitados. Como os níveis de instrumentação podem ser selecionados pelo usuário em qualquer lugar do código fonte, é possível que o usuário exerça controle sobre o nível de intrusão que a instrumentação causará em seu programa.

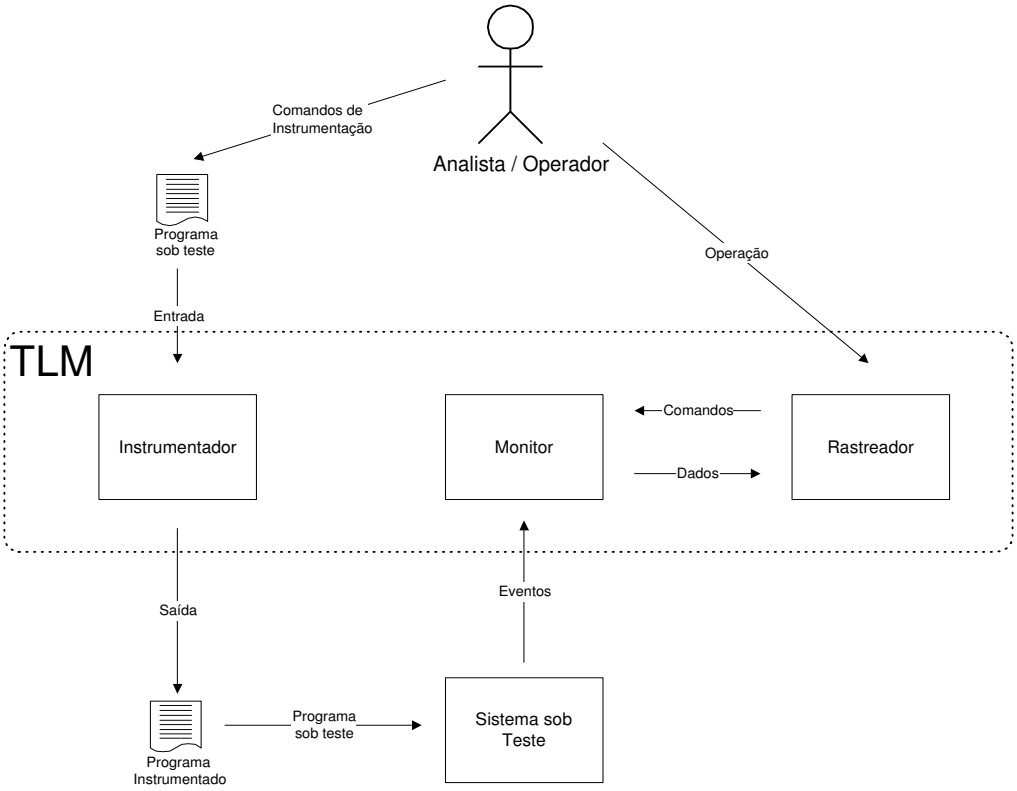


Figura 6: Diagrama de contexto do TLM.

Fonte: [Braga 99a]

O TLM é baseado em um monitor híbrido, e as instruções de instrumentação inseridas pelo instrumentador no código do usuário escrevem em uma faixa de endereços pré-determinada da memória do SUT. Fica a cargo do monitor a captura destas escritas, a anotação do instante de ocorrência de cada evento, e o armazenamento destas informações em sua memória local. As informações coletadas permanecem na memória do monitor até que o rastreador envie um comando de leitura dos dados. Todas as tarefas do monitor do TLM são efetuadas em *hardware*, sendo que a profundidade da memória de armazenamento de eventos pode ser considerada sua maior limitação.

Além de instruções de instrumentação, o monitor do TLM também é capaz de identificar a ocorrência de eventos de *hardware*, tais como interrupções, pedidos de DMA (*Direct Memory Access*) e mudança no nível lógico em sinais digitais do circuito do SUT. Para tornar possível a obtenção deste tipo de informação foi necessário dedicar o monitor para uso com o processador Intel 80C186EC. O acoplamento entre monitor e SUT é feito através de uma interface padronizada, que é semelhante àquela utilizada pelos emuladores deste modelo de processador. A adaptação de um monitor com estas características para outra arquitetura de processadores exigiria um novo desenvolvimento de *hardware* para o monitor.

Uma vez que o monitor tenha realizado capturas e que sua memória esteja lotada de eventos, as informações coletadas precisam ser transferidas para um computador para serem analisadas. No TLM esta transferência é feita pela porta paralela de um computador pessoal. A partir daí, o programa de rastreamento filtra e analisa as informações, gerando quatro formas de visualização do sistema monitorado: linhas de tempo, grafos de segmentos, histogramas e código fonte.

Com as linhas de tempo o usuário pode visualizar os eventos importantes que ocorreram no nível das tarefas e das trocas de mensagens entre as tarefas. Com os grafos de segmentos é possível saber quais partes do código foram cobertas pelo teste realizado. Os histogramas apresentam o tempo de processamento de cada tarefa, bem como o tempo em que o processador ficou ocioso. A visualização do código fonte é disponibilizada como uma forma de o usuário poder acompanhar e correlacionar o seu código com as linhas de tempo e com os grafos de segmentos. Um exemplo da interface gráfica utilizada pelo rastreador para a apresentação dos resultados no TLM é mostrado na Figura 7.

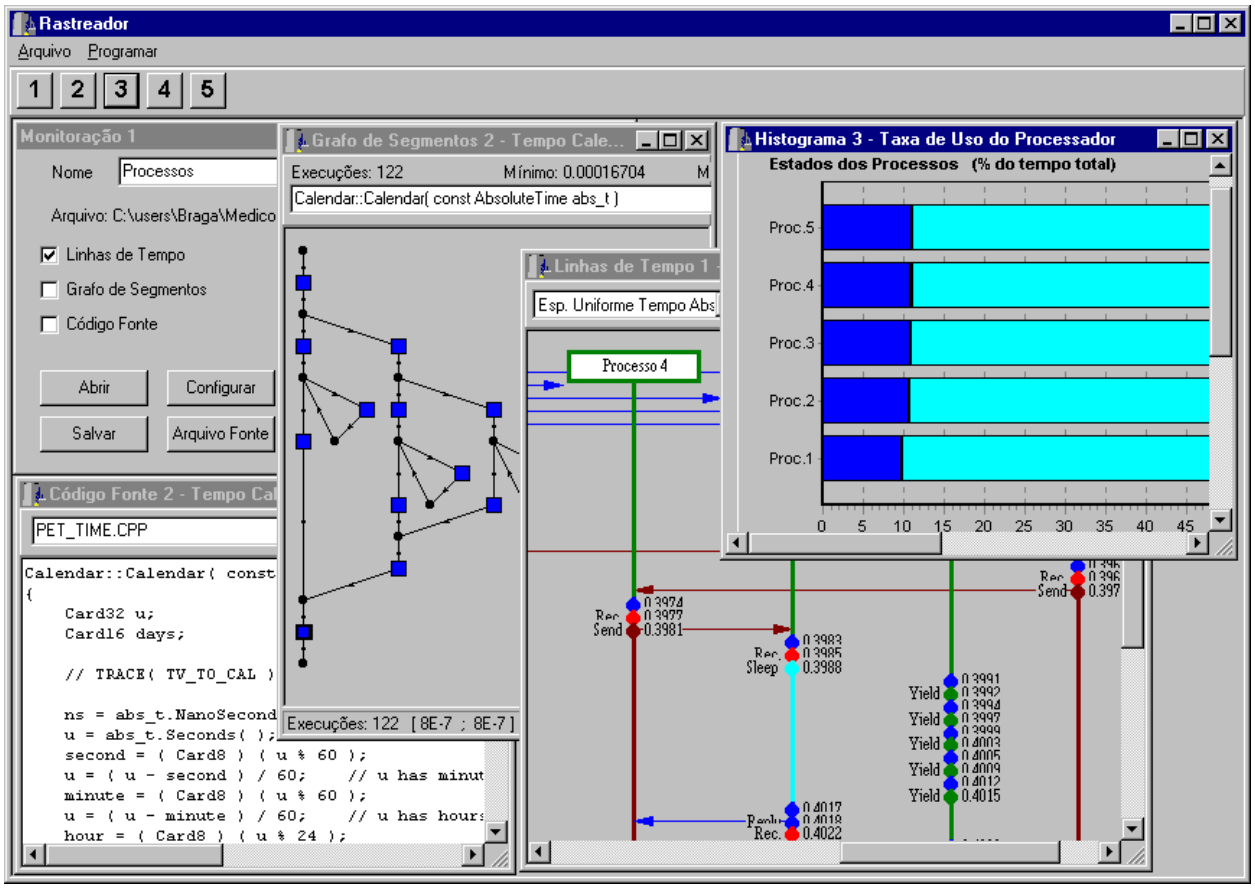


Figura 7: Interface gráfica do rastreador do TLM.

Fonte: [Braga 99a]

Em resumo, o TLM é uma ferramenta de teste em sistemas embarcados que apresenta baixa intrusão, resultante do uso de um monitor híbrido, bom suporte a testes estruturais, e formas de visualização que auxiliam na identificação de falhas estruturais e temporais. Por outro lado, o monitor híbrido do TLM não é facilmente portável para outras arquiteturas de processadores e memória do monitor para a captura de eventos é limitada.

4.4.2 VDS

O VDS (*Visualization and Debugging System*) é um sistema de visualização e depuração que, utilizando métodos dinâmicos, ajuda a identificar problemas relacionados à violação das restrições temporais em sistemas distribuídos de tempo real [Tsai 96]. Cada nó da rede no modelo do VDS executa um programa que é rastreado por um monitor baseado em *hardware*, e que se comporta como um sistema independente conectado em uma rede. Com os dados capturados pelos monitores nos diversos nós, o VDS é capaz de fazer a correlação entre todos eles e reprisar a execução do sistema para que o analista identifique os possíveis pontos de comportamento não conforme, sendo inclusive capaz de simular o comportamento de qualquer falha que tenha ocorrido.

O monitor baseado em *hardware* utilizado pelo VDS foi construído por Tsai e outros [Tsai 90] com o objetivo de monitorar processadores Motorola 68000 que utilizam sistema operacional UNIX. O monitor possui quatro blocos principais: Unidade de Controle da Interface, Unidade Duplamente Processada, Unidade de Qualificação e Memória de Alta Velocidade. Uma ilustração deste sistema de monitoração é mostrada na Figura 8.

O monitor é ligado ao SUT através da Unidade de Controle da Interface, que acompanha as atividades do processador principal observando os barramentos de endereços, dados e controle, sem interferir no sistema. A Unidade Duplamente Processada é equipada com dois processadores iguais ao do SUT. Eles são responsáveis por reproduzir as ações do processador principal a partir do ponto inicial de amostragem. As condições de início e de fim de amostragem são determinadas e processadas pela Unidade de Qualificação, enquanto a Memória de Alta Velocidade registra os eventos ocorridos nos barramentos do processador principal para posterior tratamento pela Unidade Duplamente processada. A monitoração é iniciada através de uma solicitação feita pela Unidade Duplamente Processada ao processador do SUT, utilizando uma interrupção de baixa prioridade. Na rotina de atendimento, o SUT e o monitor sincronizam seus estados para que, a partir daquele ponto, ambos comecem a executar em paralelo o mesmo código, com os mesmos valores de

variáveis, obtendo os mesmos resultados. É fato que o processo de sincronização é intrusivo, mas a partir daí o SUT não sofrerá mais interferência do Módulo de Interface.

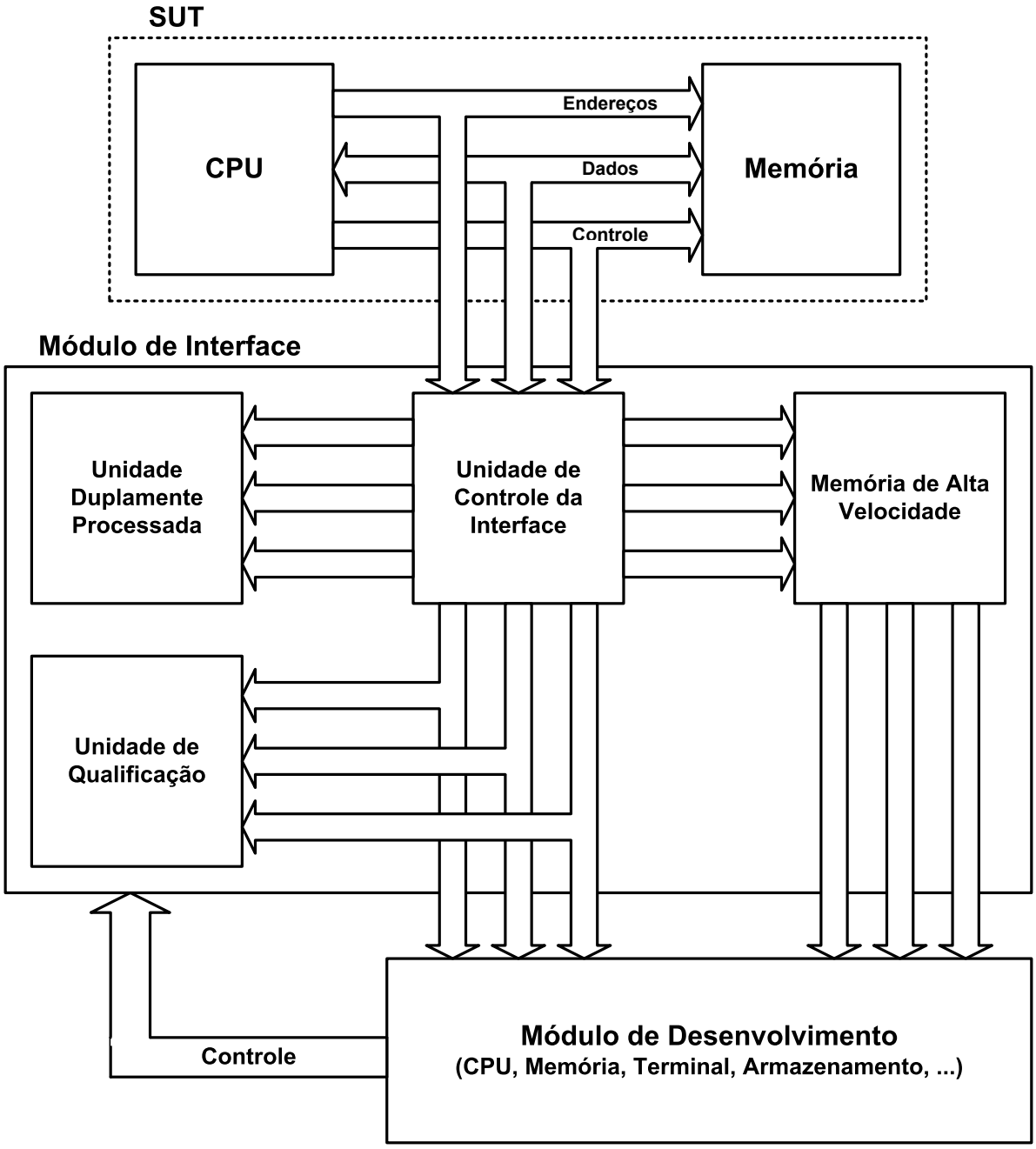


Figura 8: Arquitetura do monitor do VDS.

Fonte: [Tsai 90]

Uma restrição temporal é definida pelo VDS como "o tempo máximo permitido entre a ocorrência de dois eventos", e quatro tipos de faltas são previstas no VDS: faltas de sincronização, faltas de computação, faltas de escalonamento e faltas de computação-

escalonamento. Uma falta de sincronização ocorre quando uma tarefa despende mais tempo do que o esperado aguardando por uma mensagem ou por um recurso. Uma falta de computação ocorre quando uma tarefa despende mais tempo do que o esperado executando. Uma falta de escalonamento ocorre quando uma tarefa relevante para o funcionamento do sistema despende mais tempo do que o esperado no estado pronto, ou seja, sem receber o processador. Finalmente, uma falta de computação-escalonamento ocorre quando uma tarefa não consegue obter o processador pelo tempo necessário, mesmo tendo recebido-o por bastante tempo.

A metodologia utilizada pelo VDS para analisar as informações de monitoração capturadas é mostrada na Figura 9. Na etapa 1 o Interpretador de Rastro filtra as informações de monitoração geradas nos diversos nós da rede, descartando as que são irrelevantes e reorganizando as úteis em um único arquivo de dados.

Na etapa 2 o Correlacionador de Eventos analisa as informações contidas no arquivo de dados filtrados e reorganiza-as na forma de relacionamentos entre as diversas tarefas que compõe o sistema. O relacionamento entre tarefas é a única forma de abstração prevista no VDS. A saída da etapa 2 é o PLEL (*Process-Level Execution Log*), que é o principal registro a ser utilizado nas fases posteriores de visualização e depuração.

Na etapa 3 o Analisador de Tempo lê o PLEL e computa as informações temporais do sistema, calculando o tempo em que cada tarefa ficou em estado pronto, executando ou esperando. O resultado destes cálculos permite encontrar as faltas que o VDS procura. A saída do Analisador de Tempo são relatórios informando os resultados da análise temporal.

Na etapa 4 o Interpretador de Visualização interpreta e reorganiza o PLEL na forma de padrões de apresentação, que contêm as estruturas de dados utilizadas para mostrar os diferentes tipos de grafos coloridos com os quais o VDS trabalha.

A quinta e última etapa é responsável por dar uma visão do comportamento dinâmico do sistema para o usuário, e é feita em três partes. O Controlador de Animação é responsável por reprisar a execução do programa de um dos nós da rede, ajudando o usuário a compreender o seu comportamento. O Controlador de Retroação faz um mapeamento entre qualquer evento mostrado nos grafos do VDS e o código fonte do usuário, facilitando a localização de faltas. Finalmente, o Controlador de Apresentação é responsável por mostrar o comportamento temporal e funcional do programa distribuído através da apresentação da interação entre todos os diferentes processos que compõe o sistema.

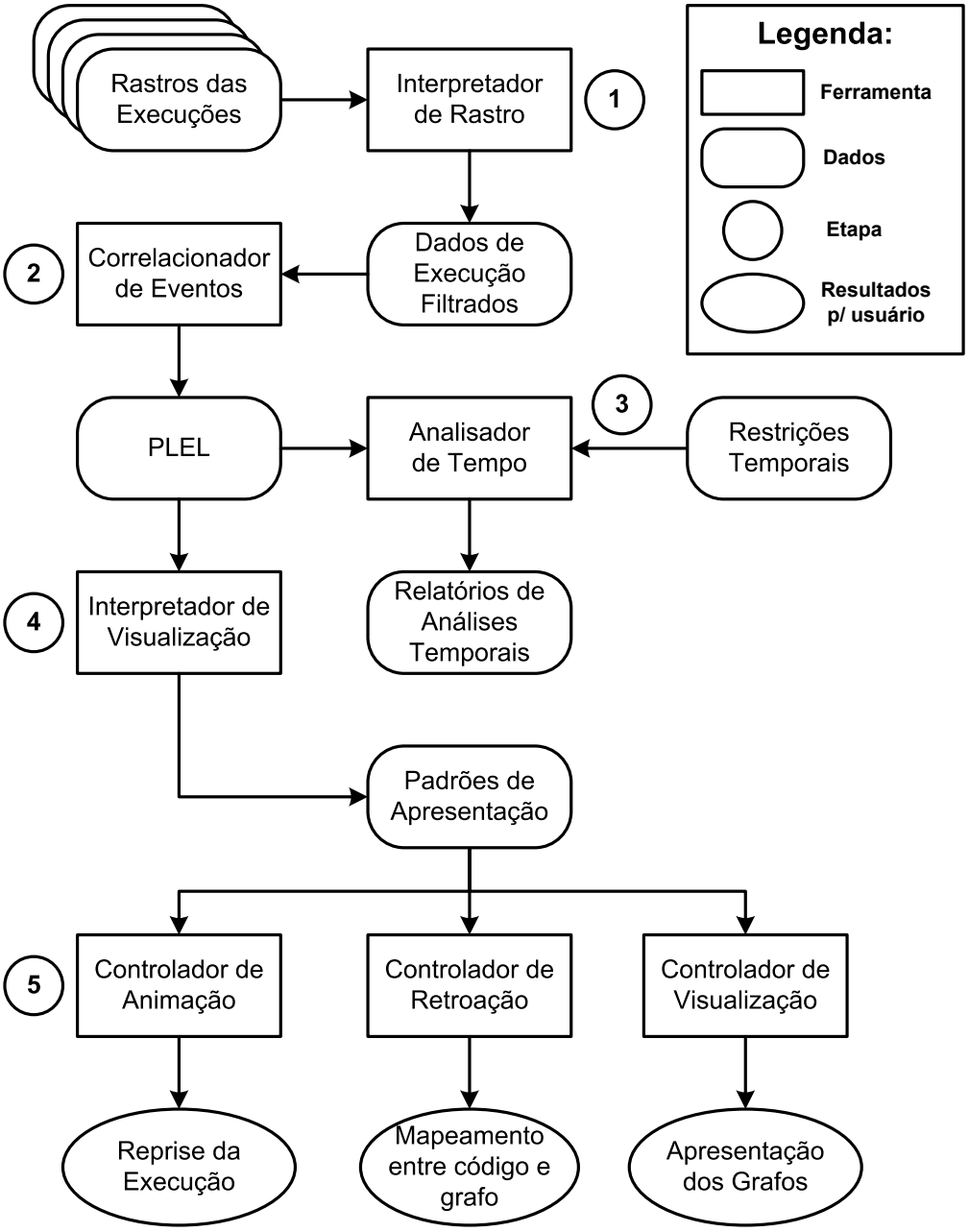


Figura 9: Metodologia do VDS.

Fonte: [Tsai 96]

Um dos objetivos do VDS é auxiliar na identificação de faltas de sincronização, e com relação a isto ele é capaz de auxiliar na identificação de seis razões para que uma falta de sincronização ocorra: *deadlock* de tarefa, término distribuído, operação faltante, inanição, perda de mensagem e *deadlock* de comunicação. Um *deadlock* de tarefa ocorre quando duas delas ficam esperando uma pela outra, sendo que nenhuma delas seja ativada. O término distribuído é um erro que ocorre quando, em um par de tarefas sincronizadas, uma delas termina e deixa a outra em um estado de espera infinita. Uma operação faltante ocorre

quando uma tarefa não se sincroniza com o restante do sistema porque, por exemplo, uma confirmação para encerramento não é enviada. A inanição é o fenômeno que ocorre quando uma tarefa de baixa prioridade não consegue prosseguir sua execução porque necessita acessar um recurso compartilhado que está sempre sendo utilizado por tarefas de mais alta prioridade. Uma perda de mensagem ocorre quando mensagens ou respostas que precisam ser trocadas para sincronizar tarefas do sistema são perdidas no caminho físico ou nas camadas inferiores do sistema operacional. Finalmente, um *deadlock* de comunicação ocorre quando uma mensagem é bloqueada na rede.

Em resumo, o VDS é um sistema que possui um método e ferramentas para auxiliar na identificação de erros dinâmicos de sistemas em tempo real distribuídos, o que é feito monitorando cada nó que compõe o sistema e reproduzindo seu comportamento posteriormente. O VDS é caracterizado pela intrusão nula no sistema monitorado, devido ao uso de um monitor baseado em *hardware*, e pela capacidade de encontrar faltas de sincronização, computação, escalonamento e computação-escalonamento em uma dada execução do sistema. No entanto, o VDS não é capaz de mostrar detalhes do sistema monitorado além do nível de trocas de mensagens entre tarefas, e é difícil portá-lo para outras arquiteturas de processadores devido ao modelo de monitoração adotado. Além disso, sua implementação depende do sistema operacional UNIX.

4.4.3 CODETEST

O CodeTEST é uma ferramenta de apoio utilizada nos testes estruturais e funcionais de programas escritos em C, C++ ou Ada, para sistemas embarcados. Com o CodeTEST é possível realizar análise de cobertura, análise de utilização do processador e análise de utilização da memória [Greenwalt 03]. Atualmente o CodeTEST é comercializado pela Freescale Semicondutores.

A arquitetura do CodeTEST é mostrada na Figura 10. Nela observa-se que o CodeTEST é baseado em instrumentação de código fonte e depende do compilador da Freescale, que é o responsável por gerar o programa executável e a base de dados com as informações sobre as instruções de instrumentação inseridas no código do usuário. O programa executável é carregado no SUT e os eventos gerados por ele são enviados para a estação de trabalho através de uma rede Ethernet. Na estação de trabalho os eventos são correlacionados com as informações contidas na base de dados, permitindo a realização das análises de cobertura, utilização do processador e utilização da memória.

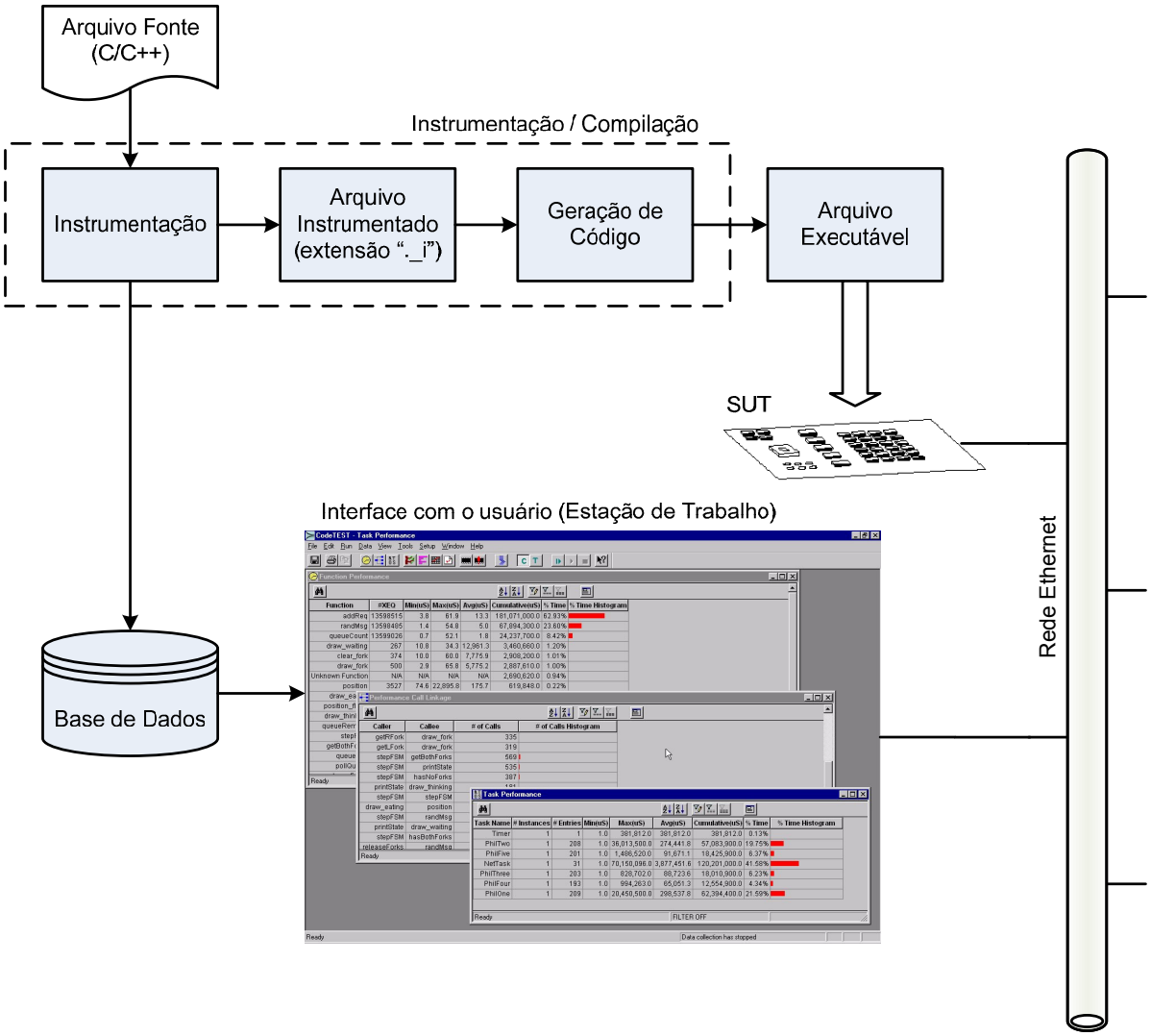


Figura 10: Arquitetura do CodeTEST.

Fonte: [Greenwalt 03]

Existem duas configurações para o CodeTEST: HWIC (*Hardware In Circuit*) e SWIC (*Software In Circuit*). O HWIC é um monitor híbrido que realiza a captura de dados através da observação dos barramentos de endereços e de dados do processador. A escrita em uma faixa de endereços pré-determinados pela IUT indica ao HWIC os dados de monitoração que devem ser capturados. Nesta configuração a responsabilidade pela comunicação com a estação de trabalho, via uma rede Ethernet, é do monitor híbrido, e não do SUT, diminuindo a intrusão causada pela monitoração. Um diagrama em blocos do funcionamento do HWIC é mostrado na Figura 11.

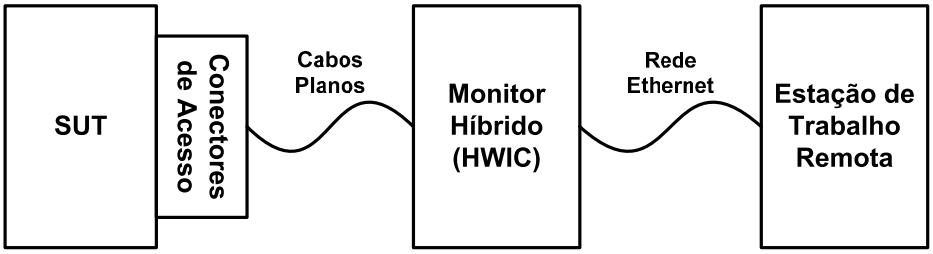


Figura 11: Diagrama em blocos do HWIC.

Fonte: [Greenwalt 03]

O SWIC é um monitor baseado em *software* que, ao contrário do HWIC, não requer um equipamento externo conectado ao SUT. O monitor do SWIC é uma biblioteca que precisa ser ligada junto com a aplicação, e que tem por função coletar as informações necessárias para a monitoração do SUT. Esta abordagem, apesar de menos dispendiosa que o HWIC, exige mais recursos do SUT, entre eles memória para armazenamento de eventos e uma porta Ethernet para envio das informações de monitoramento para a estação de trabalho, além de ser mais intrusiva.

Uma comparação entre as abordagens de teste utilizando o HWIC e o SWIC do CodeTEST, conforme apresentada em [Greenwalt 03], é mostrada na Tabela 2.

Tabela 2: Comparação entre HWIC e SWIC.

Característica	SWIC	HWIC
Interação com o SUT	Agente no SUT	Monitor Híbrido Externo
Acurácia	Baixa	Alta
Intrusão	Alta	Baixa
Portabilidade para um RTOS	Baixa	Alta
Facilidade para configurar	Simples	Complexa
Custo	Menor	Maior

Em resumo, o CodeTEST é uma ferramenta utilizada em testes estruturais e funcionais, e que a princípio não está preparada para fazer a avaliação temporal de ERTS. Apesar disso, o HWIC é um monitor híbrido de baixa intrusão e alta acurácia, e que poderia ser utilizado para realizar a coleta de dados necessária para que uma ferramenta externa fizesse algum tipo de avaliação temporal. Entretanto, esta possibilidade não está disponível no momento porque a tecnologia envolvida é um segredo industrial embutido no HWIC, nos pré-compiladores e nos compiladores da Freescale.

4.4.4 WINDVIEW

O WindView [Wind 99] é uma ferramenta visualização do comportamento dinâmico de um SUT, cujo conceito foi apresentado inicialmente por Wilner em 1995 [Wilner 95]. O WindView está ligado ao sistema operacional VxWorks [Wind 95], e através dele o usuário é capaz de observar o comportamento das tarefas que executam no SUT, incluindo:

- Estados das tarefas.
- Comutações de contexto.
- Interações entre tarefas.
- Interferências externas, como o atendimento a interrupções.
- Envio e recebimento de mensagens.
- Atuações do escalonador.
- Utilização de semáforos.
- Taxa de utilização do processador por tarefa.
- Dados estatísticos sobre a execução das tarefas durante a monitoração feita.

O WindView utiliza um monitor baseado em *software* cuja instrumentação encontra-se concentrada no sistema operacional. O monitor fica em estado latente até que seja iniciada uma captura de eventos, o que pode ocorrer por um pedido explícito do usuário ou em virtude de o programa ter atingido um determinado estado previamente marcado como início de amostragem (*trigger*). Quando a monitoração é iniciada, o monitor aloca memória do sistema e começa a armazenar nela os eventos capturados. O armazenamento de eventos no SUT é a forma encontrada para minimizar a utilização de banda na comunicação entre o programa de rastreamento e a plataforma alvo. Nesta forma de operação, os dados coletados permanecem no SUT até que a monitoração termine ou até que a memória alocada pelo monitor fique lotada de eventos. Neste último caso o agente de monitoração envia os dados coletados até aquele momento para a estação de trabalho, permitindo que a monitoração continue. Ao término da monitoração, o programa de análise, que executa na estação de trabalho, processa todos os dados capturados e apresenta-os ao usuário na forma de um gráfico de Gannt, conforme mostrado na Figura 12.

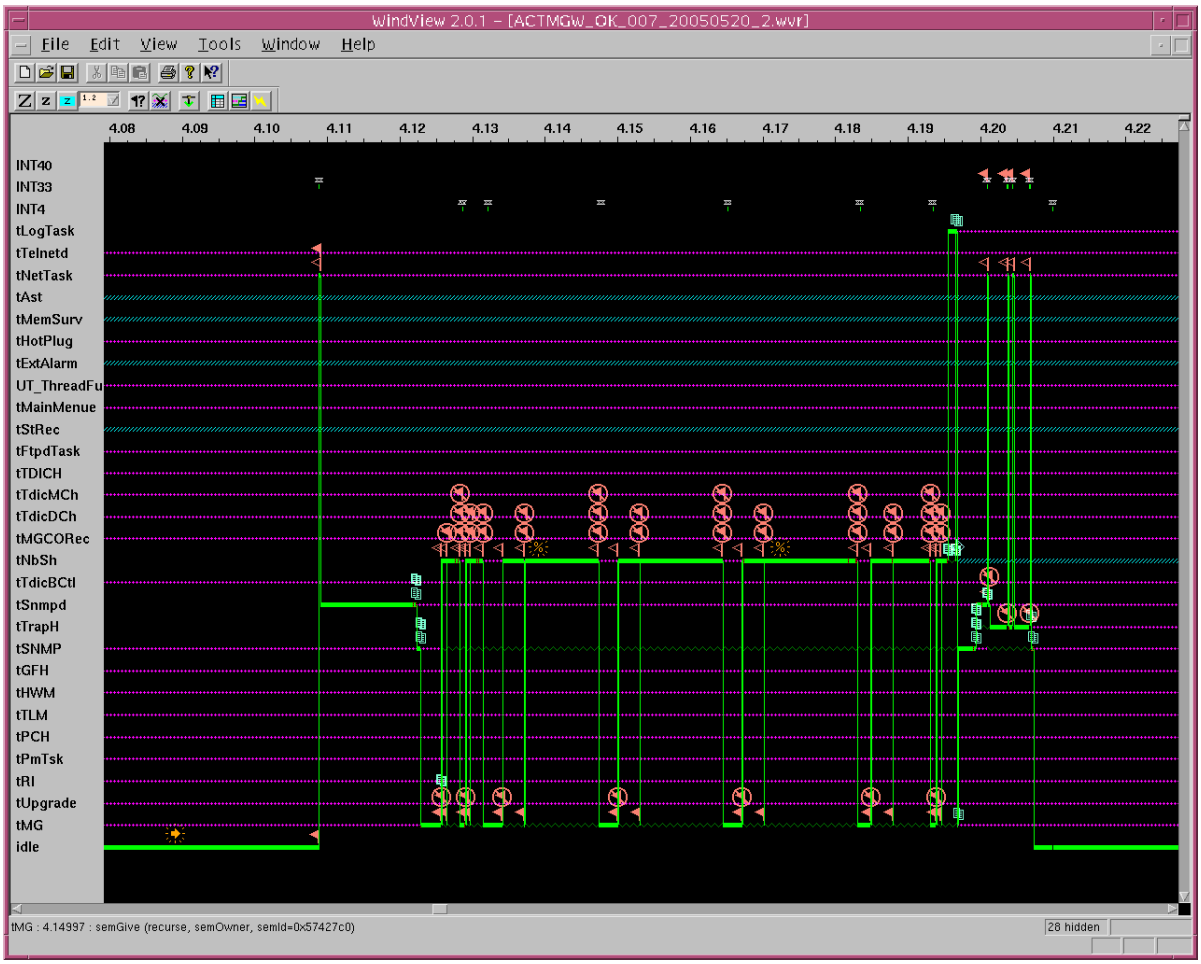


Figura 12: Apresentação de resultados por gráfico de Gannt no WindView.

Em resumo, o WindView é uma ferramenta de visualização do comportamento dinâmico de um sistema baseado no sistema operacional VxWorks. Por ser um monitor baseado em *software*, o WindView independe da arquitetura do processador utilizado, sendo capaz de mostrar tanto eventos do sistema operacional quanto alguns eventos do usuário, utilizando para isto uma interface de *software* própria. No entanto, o WindView depende do sistema operacional VxWorks, podendo alterar a ordem de execução do sistema quando ativado e, por isso, é passível de gerar o efeito sonda.

Ferramentas semelhantes ao WindView também foram desenvolvidas para outros sistemas operacionais. Um exemplo é o LTT (Linux Trace Toolkit) [Opersys 07], desenvolvido para Linux. Por se tratar de *software* livre, o LTT foi adaptado posteriormente para uso em sistemas embarcados operando em tempo real que não utilizavam o Linux [Bregant 05].

4.4.5 REPRISE UTILIZANDO MÁQUINAS DE TEMPO

A Reprise Utilizando Máquinas de Tempo é um método desenvolvido pelo Centro de Pesquisas em Sistemas de Tempo Real da Universidade de Mälardalen, na Suécia, com o objetivo de diagnosticar as causas de falhas intermitentes causadas por problemas de concorrência ou pela violação de restrições temporais. Ele consiste em estender a técnica da reprise, apresentada originalmente por LeBlanc e Mellor-Crummey [LeBlanc 87], para sistemas distribuídos operando em tempo real.

A reprise é uma forma de depuração baseada em monitoração que armazena os eventos significativos que ocorrem durante a execução do sistema. Uma vez de posse destes eventos, deve ser possível reproduzir a posteriori o comportamento do sistema no mesmo cenário em que o registro ocorreu. A reprodução pode ser tão detalhada quanto o registro permitir. O objetivo desta técnica é rastrear o sistema em busca de uma falha específica, normalmente de natureza intermitente, não sendo adequada para explorar o comportamento geral do sistema.

O trabalho de Thane [Thane 00] apresentou alguns avanços em relação a outras propostas surgidas até então. Entre estes avanços podemos destacar o uso de sistemas operacionais tempo de real e ferramentas comerciais de desenvolvimento, além da aplicabilidade em ambientes multitarefa e distribuídos. Na solução proposta, foi adotada a monitoração baseada em *software* e, para efetividade dos resultados, algumas restrições precisavam ser observadas. A mais relevante delas exigia que existisse um simulador do sistema operacional preparado para reproduzir os eventos registrados durante a monitoração. Outra restrição exigia que o RTOS utilizado permitisse o registro pelo sistema de monitoração de eventos como interrupções, preempções e comutações de contexto.

Três anos mais tarde, a mesma equipe apresentou o conceito de Máquina de Tempo [Thane 03], que melhorou o método anterior, removendo a necessidade de haver um simulador do sistema operacional. Com as idéias propostas, a reprise do sistema passou a ser feita na mesma plataforma em que o programa havia sido executado originalmente, desde que a depuração fosse feita remotamente. A Máquina de Tempo é composta por três elementos: um Gravador, um Historiador e um Viajante do Tempo.

- Gravador: monitor baseado em *software* que executa no SUT, juntamente com a aplicação, e que coleta todas as informações necessárias a respeito do fluxo de dados, do fluxo de controle, das comutações de contexto e das interrupções.

- Historiador: programa que executa em uma estação de trabalho, portanto fora do SUT, para realizar a análise dos registros gerados durante a execução, e também para correlacioná-los no tempo. O Historiador gera como saída um conjunto de macros que alimenta um depurador com as informações sobre onde colocar pontos de paradas condicionais para que seja possível executar novamente o programa com as informações colhidas a partir do registro. Também é responsabilidade do Historiador gerar a lista de predicados relacionados a cada ponto de parada condicional.
- Viajante do Tempo: É um programa que interage com o depurador, permitindo que as informações coletadas pelo Historiador possam ser passadas para uma seção do programa em execução (ex.: variáveis de estado, variáveis globais, contador de programa, etc...). Esta interação permite a recriação do estado do programa em qualquer instante de tempo que esteja no escopo do Historiador.

Junto com o trabalho da Máquina de Tempo a equipe de pesquisadores apresentou também uma solução para o problema de se determinar um ponto para o reinício da execução de um programa [Huselius 03]. Isto foi necessário porque o registro coletado pelo Gravador contém apenas as atividades recentes do programa, e não um histórico completo desde o início da operação. Para que o reinício seja possível existem dez pré-requisitos que precisam ser atendidos. Entre eles podemos destacar a necessidade um registro detalhado tanto do fluxo de controle como do fluxo de dados, além da inexistência de canais de comunicação que não possam ser reproduzidos.

Em resumo, a Reprise Utilizando Máquinas de Tempo é um método desenvolvido para diagnosticar problemas intermitentes, causados pela concorrência ou pela violação das restrições temporais, que oferece a flexibilidade da monitoração baseada em *software* e a possibilidade de uso com ferramentas comerciais. Por outro lado, o sistema monitorado sofre um alto nível de intrusão, e precisa prever processamento e espaço de armazenamento para a monitoração, uma vez que sua remoção implicaria o efeito sonda.

4.5 RESUMO

Este capítulo descreveu o ambiente de trabalho e revisou a bibliografia que envolve a verificação temporal de sistemas embarcados que operam em tempo real. A descrição do ambiente de trabalho apresentou as características da plataforma de *hardware* embarcada, do ambiente de programação e do ambiente de execução. No ambiente encontram-se algumas das condições de contorno que permitem delimitar o escopo do problema e determinar a abrangência do método proposto.

A revisão bibliográfica citou cinco trabalhos relacionados: o TLM, o VDS, o CodeTEST, o WindView e a Reprise Utilizando Máquinas de Tempo. A Tabela 3 mostra uma comparação das características principais destes cinco trabalhos.

Tabela 3: Comparação entre os métodos e ferramentas estudados.

Método e/ou Ferramenta		Tipo de Monitoração	Independência da arquitetura do Processador	Suporte a diferentes RTOS	Independência do ambiente de desenvolvimento	Instrumen-tação Seletiva	Intrusão	Testes Estrutu-rais
TLM		Híbrida	Não	Não	Sim	Sim	Baixa	Sim
VDS		Hardware	Não	Não	Sim	Não	Nula	Não
Code TEST	HWIC	Híbrida	Não	Sim	Não	Não	Baixa	Sim
	SWIC	Software	Sim	Não	Não	Não	Alta	Sim
WindView		Software	Sim	Não	Não	Não	Alta	Não
Reprise		Software	Sim	Sim	Sim	Não	Alta	Não

O próximo capítulo apresenta o método Dyretiva, cujo objetivo é definir um processo que auxilie na verificação das restrições temporais de ERTS por métodos dinâmicos.

5 O MÉTODO DYRETIVA

O Dyretiva é um método desenvolvido para atender a algumas das necessidades de testes em sistemas embarcados que operam em tempo real, incluindo a verificação de restrições temporais. O Dyretiva é baseado em monitoração (Capítulo 3) para obter informações sobre o comportamento do sistema, ou seja, utiliza um método dinâmico (Seção 2.3.2) na determinação de tempos de execução.

No processo de desenvolvimento, a fase de testes está ligada à avaliação da qualidade do *software* embarcado (Seção 2.1), que é essencial em ERTS. Porém, a fase de testes precisa considerar tanto os aspectos funcionais quanto os aspectos temporais. Em virtude disto, são introduzidos alguns problemas decorrentes da necessidade de avaliação temporal no sistema, que resultam principalmente da interação que existe entre o ERTS e o ambiente, da influência da programação concorrente no comportamento temporal das tarefas, e da possibilidade de introduzir o efeito sonda em conseqüência da observação (Seção 2.2.2).

Neste capítulo são apresentados, inicialmente, os requisitos do método Dyretiva e o seu modelo de utilização, seguidos da descrição de seus dois principais elementos: a abordagem de monitoração e o modelo de falta. A abordagem de monitoração padroniza os recursos de teste e define a forma de integração com diferentes RTOS. O modelo de falta define as circunstâncias em que o Dyretiva espera encontrar uma falta.

5.1 REQUISITOS

O Dyretiva é um método utilizado durante a fase de testes de sistemas embarcados operando em tempo real, e a coleta de dados é feita através da monitoração do SUT. Porém, a fase de testes em ERTS não envolve somente a verificação temporal do sistema. Por este motivo, o Dyretiva prevê que os recursos disponibilizados para teste no sistema embarcado possam ser utilizados tanto na verificação temporal quanto na lógica. Por isso, a monitoração no Dyretiva possui dois objetivos diferentes: monitoração para teste e depuração, e monitoração para correção e avaliação de desempenho (Seção 3.2). A monitoração para teste e depuração é utilizada durante os testes de unidade e de integração (Seção 2.2.1), bem como durante os testes de tarefas (Seção 2.2.2). Já a monitoração para correção e avaliação de desempenho é utilizada durante os testes de comportamento, e de comunicação e sincronização de tarefas (Seção 2.2.2), bem como durante o teste de sistema (Seção 2.2.1). A Figura 13 ilustra como é a fase de testes em um processo que utiliza o Dyretiva.

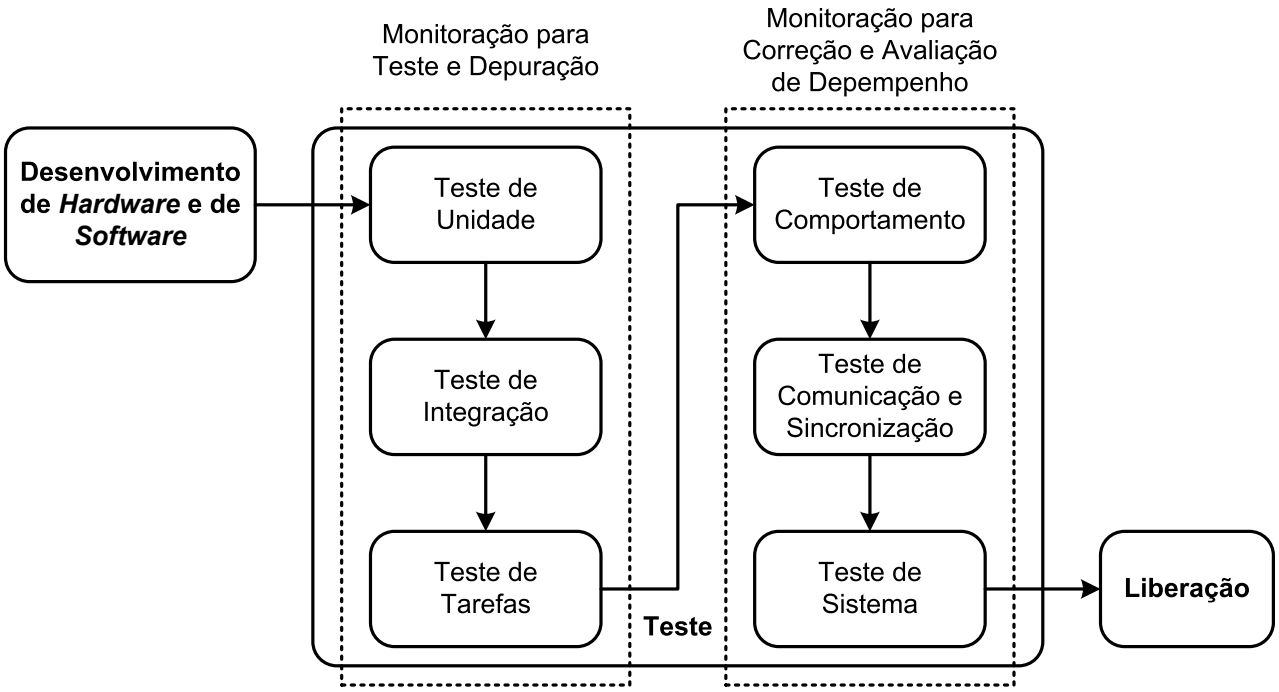


Figura 13: Detalhamento da fase de testes para o Dyretiva.

Um dos objetivos do Dyretiva é atender plataformas de *hardware* que utilizam processadores de 16 ou de 32 bits (Seção 4.1). Para atingir este objetivo, uma das características esperadas do método é portabilidade para diferentes arquiteturas de processadores.

Sendo baseado em monitoração, o Dyretiva deve utilizar um dos métodos de monitoração descritos na Seção 3.3. Em virtude da área de aplicação ser sistemas embarcados que podem possuir recursos restritos (Capítulo 2), verifica-se que a monitoração baseada em *software* não é adequada, pois podem ser necessários recursos de memória e de processamento não disponíveis no sistema embarcado, além deste tipo de monitoração ser muito intrusiva. A monitoração baseada em *hardware* também não é adequada, pois sua utilização em processadores modernos com muitos estágios de *pipeline* seria muito complexa e não seria portável. Assim sendo, o método mais apropriado neste caso é a monitoração híbrida, tanto em termos de portabilidade, quanto em termos de minimização da intrusão causada no SUT. Apesar das qualidades para a aplicação em ERTS, a monitoração híbrida não elimina a intrusão do sistema monitorado, podendo mudar o comportamento temporal do sistema e introduzir o efeito sonda (Seção 2.2.2). Em virtude disto, este quesito deve ter atenção especial por parte do método Dyretiva.

A monitoração de um sistema com o objetivo de verificar as restrições temporais precisa de um nível de detalhe que requer que tanto a aplicação quanto o RTOS sejam instrumentados. A aplicação é um programa que normalmente possui muitos pontos de decisão que poderiam ser observados externamente. Se a instrumentação da aplicação ficar a cargo somente do usuário, provavelmente será muito demorada e sujeita a erros. Por isto, esta tarefa deve ser delegada para um instrumentador automático de código fonte. No entanto, os instrumentadores automáticos usualmente colocam instruções de instrumentação em todos os pontos de decisão do programa, o que em muitos casos não agrega valor para o rastreamento e pode sobrecarregar os recursos do sistema. Além disso, a instrumentação de um pequeno subconjunto de todos os possíveis pontos de decisão pode atender as necessidades de realizar avaliação temporal. Por este motivo, o Dyretiva deve prever uma política de instrumentação que proporcione o apoio necessário para que os usuários possam instrumentar seu código fonte, mas que ao mesmo tempo dêem a ele a possibilidade de selecionar que partes do código devem ser instrumentadas ou não, minimizando a intrusão. Esta política de instrumentação fica sujeita a limitação imposta na Seção 4.2, que restringe a linguagem de programação ao C. Apesar disso, o conceito pode ser estendido para outras linguagens de programação, incluindo o C++.

Ao contrário da aplicação, o RTOS não pode ser instrumentado por um instrumentador automático, pois não basta apenas sinalizar a ocorrência de eventos. É necessário detalhar o evento informando, por exemplo, o remetente e destinatário quando uma mensagem é enviada. Somente com estas informações é que o sistema de rastreamento será capaz de recuperar posteriormente as linhas de execução das tarefas. Por este motivo, a instrumentação do sistema operacional pode precisar de formatos específicos de instruções de instrumentação que permitam utilizá-la para atender a diferentes necessidades de sinalização de eventos dos sistemas operacionais. Além disso, como o RTOS precisa sinalizar eventos contendo mais informações, a inclusão de pontos de instrumentação precisa ser feita com muito critério, de preferência por especialistas, com o intuito de minimizar os pontos de inserção e, em conseqüência, a intrusão causada. Com isto, esta tarefa passa a depender de suporte do fornecedor ou da disponibilidade do código fonte, que deverá ser estudado em detalhes para inserir a instrumentação necessária.

A monitoração de um sistema retorna uma grande quantidade de dados, referentes a eventos do usuário e do sistema operacional, que sem algum processo de filtragem, análise e apresentação não têm significado algum para um observador externo. Em virtude disto, é necessário associar o método a um conjunto de ferramentas que sejam capazes de extrair informações a partir dos dados coletados, e apresentar estas informações para o usuário de modo a facilitar a identificação de problemas e otimizar a processo de teste.

Em resumo, esta seção mostrou os requisitos do método Dyretiva, que são: utilização tanto em testes funcionais quanto na verificação das restrições temporais; portabilidade para diferentes arquiteturas de processadores; monitoração híbrida utilizando poucos recursos da plataforma de *hardware* e com imunidade ao efeito sonda; instrumentação automática e seletiva da aplicação; instrumentação eficiente do sistema operacional; associação com um conjunto de ferramentas que facilitem a obtenção de informações por observadores externos. Estes requisitos foram definidos levando em consideração as necessidades de teste em sistemas embarcados operando em tempo real.

5.2 MODELO DE UTILIZAÇÃO

O modelo de utilização do Dyretiva é mostrado na Figura 14, e resume como o método será utilizado na fase testes de um sistema embarcado. A fase de testes inicia-se ao final do desenvolvimento de *hardware* e de *software*, como está mostrado na Figura 1. O resultado do desenvolvimento de *hardware* é um módulo microprocessado, enquanto o resultado do desenvolvimento de *software* é o código fonte, escrito em linguagem C, que quando compilado gera o código executável. Para que os testes possam ter início, o código do usuário precisa ser instrumentado e uma base de dados com as informações da instrumentação inserida no código precisa ser gerada. Esta base de dados será utilizada posteriormente para identificar os eventos do programa do usuário durante o rastreamento. No Dyretiva a instrumentação e a base de dados são feitas por um instrumentador de código. Uma vez que o código do usuário tenha sido instrumentado, ele é ligado com o sistema operacional, que também já deve ter sido instrumentado previamente por especialistas, e com isto a aplicação está pronta para ser testada na plataforma de *hardware*.

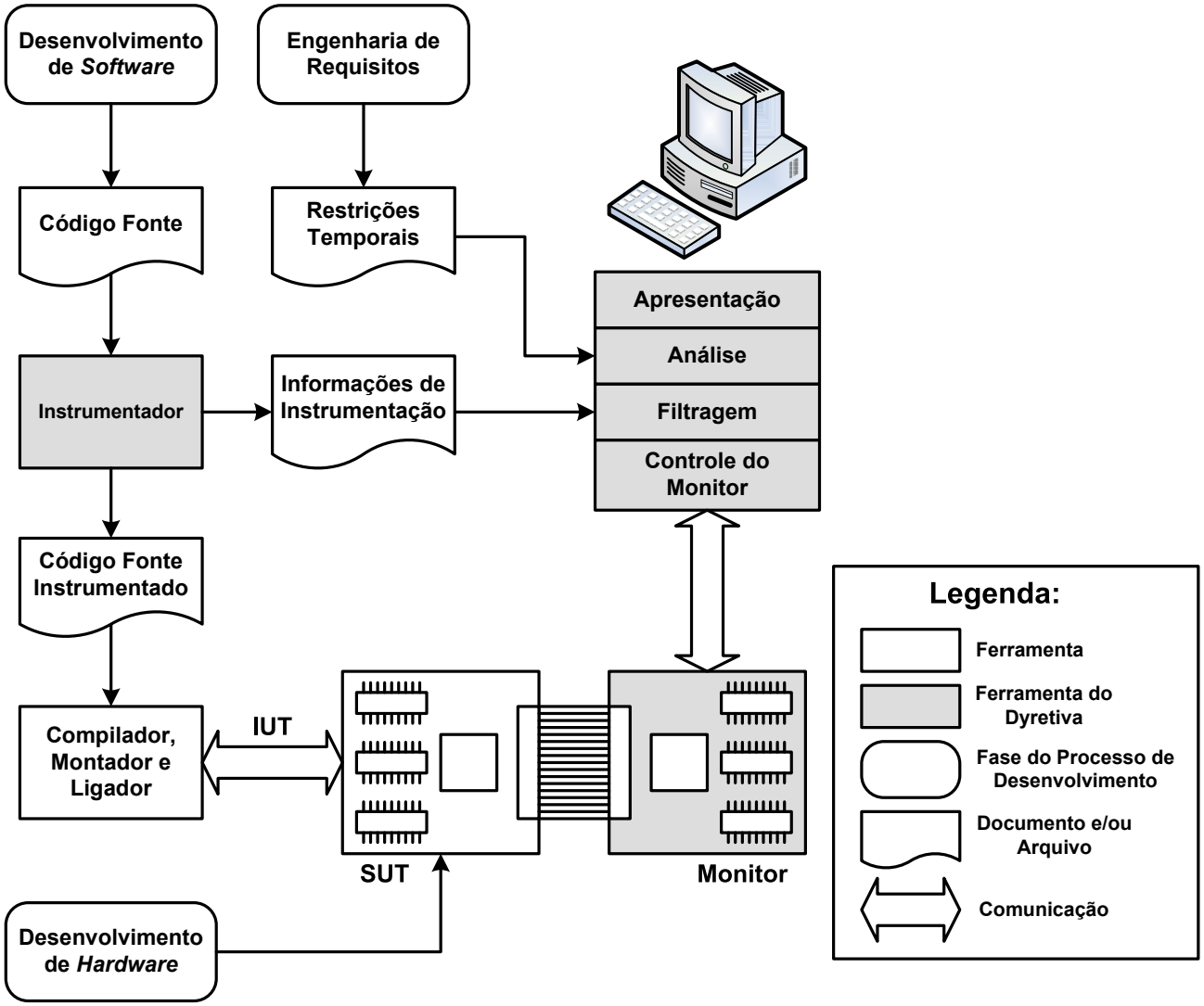


Figura 14: Modelo de utilização do Dyretiva.

O passo seguinte no modelo de utilização é a execução monitorada do programa na plataforma de *hardware*. Devido ao modelo de monitoração híbrida adotado, o monitor é externo ao SUT e responsável por capturar os eventos sinalizados. Além de realizar a captura, o monitor realiza com seus próprios recursos tarefas de monitoração que consumiriam muito tempo e recursos do sistema embarcado, tais como marcar o tempo do evento, armazená-lo em memória e comunicar-se com o observador externo.

O controle do monitor é feito pela estação de trabalho do observador através de um programa de controle do monitor. Com este programa o observador pode iniciar ou encerrar uma monitoração, bem como receber os dados capturados durante uma determinada observação do sistema. A comunicação entre o monitor e o programa é feita em um meio físico padronizado, utilizando um protocolo de comunicação definido.

Uma vez que o programa de controle do monitor tenha coletado os dados de monitoração, programas de filtragem e de análise são acionados para extrair dos dados as informações procuradas. Durante a monitoração com o objetivo de teste e depuração, os dados são correlacionados com as informações de instrumentação adicionadas ao código do usuário pelo instrumentador automático, e podem gerar informações sobre a cobertura do teste realizado. Durante a monitoração com o objetivo de correção e avaliação de desempenho, são geradas informações sobre o comportamento dinâmico do sistema, tais como a taxa de utilização do processador pelo sistema operacional e pelas das tarefas, as linhas de execução das tarefas no tempo, os tempos gastos com algoritmos críticos, e eventuais violações das restrições temporais.

Após a filtragem e a análise dos dados, formas de apresentação apropriadas para cada resultado obtido são necessárias. Janelas mostram dados informativos sobre as propriedades do sistema monitorado e dos arquivos de rastreamento. Tabelas mostram o tempo de execução de funções, tarefas, serviços do sistema operacional, exceções, rotinas de atendimento e eventuais violações das restrições temporais analisadas. Gráficos de setores mostram a ocupação do processador pelas tarefas e pelo sistema, enquanto o comportamento dinâmico do sistema é apresentado em um gráfico da Gannt.

Em função dos resultados obtidos durante uma determinada monitoração do sistema, o observador pode seguir executando um novo caso de teste, ou então reportar as condições em que uma falha foi encontrada para que seja analisada e corrigida pelas equipes de desenvolvimento de *hardware* ou de *software*.

5.3 ABORDAGEM DE MONITORAÇÃO

A abordagem de monitoração do Dyretiva foi desenvolvida para atender aos seus requisitos através da padronização física e lógica das interfaces de teste. Esta padronização compreende seis aspectos: a interface entre SUT e monitor; as classes de instrumentação; a instrumentação seletiva; o formato das instruções de instrumentação; a lógica de temporização de eventos; e o processo de instrumentação do código do usuário.

5.3.1 INTERFACE FÍSICA ENTRE SUT E MONITOR

A interface física entre o SUT e o monitor foi padronizada com 16 sinais de dados digitais, um sinal de controle e uma linha de referência de terra. Tanto o sinal de controle como os sinais de dados são saídas para o sistema embarcado e entradas para o monitor. A

lógica de funcionamento é simples: a ocorrência de uma borda de descida no sinal de controle avisa ao monitor que um novo evento encontra-se disponível nas 16 linhas de dados. Esta interface é chamada de porta de medição, e sua configuração atende a um grande número de sistemas embarcados equipados com processadores de 16 ou de 32 bits. A implementação da porta de medição torna o sistema independente da arquitetura do processador e requer poucos recursos da plataforma de *hardware*.

A implementação sugerida para a porta de medição é uma posição de memória de 16-bits. Para separar o barramento de dados do processador da conexão com a porta de medição é necessário utilizar um *latch*, conforme mostrado no esquema da Figura 15. Caso o processador não disponha de barramento de dados externo com pelo menos 16-bits de largura, ainda assim é possível implementar a porta de medição utilizando, por exemplo, portas de saída disponíveis no processador. Neste caso, porém, a escrita na porta de medição tende a ser mais demorada do que a escrita em uma posição de memória, fazendo com que a atividade de monitoração seja mais intrusiva.

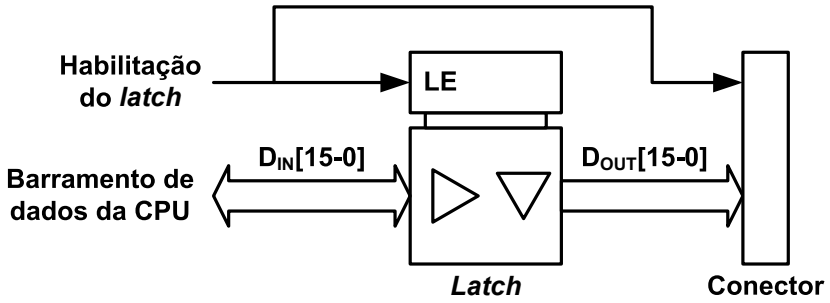


Figura 15: Esquema da implementação sugerida da porta de medição no SUT.

5.3.2 CLASSES DE INSTRUMENTAÇÃO

As instruções de instrumentação podem ser classificadas segundo sua finalidade. Quando inseridas no código do usuário, o Dyretiva divide as instruções de instrumentação em três classes: instrumentação estrutural, instrumentação contextual e instrumentação especulativa. Quando inseridas no código do sistema operacional as instruções de instrumentação são classificadas de acordo com a organização interna e com os serviços oferecidos.

A instrumentação estrutural está associada a eventos estruturais do programa, tais como início e final de função, tomadas de decisão (**`if/then/else/switch/case`**), ou estruturas de repetição (**`do/while/for`**). Para descrever um evento estrutural é necessário apenas um identificador de evento. Ao ser solicitado, o instrumentador de código insere uma chamada de função na forma **`{instr(event_id);}`** em cada ponto onde a instrumentação estrutural é necessária no programa monitorado. O protótipo da função (ou macro) **`instr`** é definido no arquivo de cabeçalho **`"instr.h"`**, cujo conteúdo depende da plataforma de *hardware*. O valor **`event_id`** deve ser único e tal que, de forma inequívoca, identifique o evento.

Quando utilizada na monitoração com o objetivo de teste e depuração, a instrumentação estrutural é capaz de obter informações sobre cobertura do teste realizado. Por outro lado, quando utilizada na monitoração com o objetivo de correção e avaliação de desempenho, a instrumentação estrutural pode informar o tempo de execução de funções ou de trechos críticos do programa.

A instrumentação contextual está associada com a marcação de variáveis do programa, permitindo que o usuário acompanhe mudanças de valores nestas variáveis. Os eventos contextuais também são descritos, assim como os eventos estruturais, por um identificador único, seguido do valor atual da variável marcada. Assim, quando o instrumentador localiza uma variável no código fonte do usuário que deve ser marcada, ele procura por todas as atribuições explícitas feitas a ela e acrescenta, após cada atribuição, uma chamada a uma das duas funções (ou macros) de instrumentação contextual: **`{dinstr16(data16_id,val16);}`** ou **`{dinstr32(data32_id,val32);}`**. A função inserida depende da largura da variável marcada ser 16 ou 32 bits, o que é reconhecido pelo instrumentador. Variáveis de 8 bits são consideradas casos especiais das variáveis de 16 bits, dada a definição da interface física entre monitor e SUT. Protótipos das funções (ou macros) **`dinstr16`** e **`dinstr32`** estão contidas no arquivo de cabeçalho **`"instr.h"`**. Uma limitação da instrumentação de variáveis é sua incapacidade de perceber as atribuições indiretas, como as que podem ser feitas através do uso de ponteiros da linguagem C.

Quando utilizada na monitoração com o objetivo de teste e depuração, a instrumentação contextual é capaz de informar a evolução em valores de variáveis do programa sem a necessidade inserir pontos de parada para coletar estes valores. Por outro lado, quando utilizada na monitoração com o objetivo de correção e avaliação de desempenho, a instrumentação contextual permite que os valores de variáveis chaves do

programa sejam rastreadas para verificar se o estado interno do sistema é consistente durante a execução de um determinado teste.

A instrumentação especulativa é utilizada para monitorar o tempo de execução de funções de biblioteca cujo código fonte não esteja disponível. Para que se possa realizar este tipo de instrumentação, o usuário deve fornecer ao instrumentador uma lista das funções a monitorar. Quando uma chamada a uma destas funções é localizada no código do usuário, dois eventos estruturais são inseridos: um de início de função logo antes da chamada, e um de retorno de função logo após a chamada. Do ponto de vista de análise do arquivo de rastreamento, não há diferença entre instrumentação estrutural de funções e a instrumentação especulativa, apesar de o instrumentador tratar ambas de forma diferente. A instrumentação especulativa possui os mesmos objetivos da instrumentação estrutural de funções, tanto no contexto da monitoração com o objetivo de teste e depuração quanto no contexto da monitoração com o objetivo de correção e avaliação de desempenho.

A instrumentação do sistema operacional não é classificada pelos mesmos critérios da instrumentação do código do usuário, mas sim conforme a organização interna e os serviços oferecidos. Entretanto, é aplicável tanto na monitoração com o objetivo de teste e depuração quanto na monitoração com o objetivo de correção e avaliação de desempenho. Durante a monitoração com o objetivo de teste e depuração, a instrumentação do sistema operacional auxilia no teste de tarefas. Posteriormente, durante a monitoração com o objetivo de correção e avaliação de desempenho, sua utilização permite obter informações sobre a taxa de utilização do processador, tempo de execução das tarefas, e o estado interno de cada tarefa. Também é pela instrumentação do sistema operacional que se torna possível reconhecer a ocorrência de eventos ligados à programação concorrente, tais como preempções, chaveamentos de tarefas, exceções, interrupções e acessos a regiões críticas.

5.3.3 SELETIVIDADE DA INSTRUMENTAÇÃO

A seletividade da instrumentação pode ser descrita como a flexibilidade dada pelo método ao usuário para que ele indique que partes do seu código devem ser instrumentadas pelo instrumentador automático.

Para a instrumentação estrutural, a seletividade é expressa em comandos de instrumentação, inseridos no código fonte pelo programador na forma de comentários de formato pré-determinado. Três comandos diferentes são reconhecidos pelo instrumentador para a instrumentação estrutural: **`/* instr_none */`**, **`/* instr_func */`** e **`/* instr_full */`**.

`/* instr_none */` instrui o instrumentador a não colocar qualquer instrução de instrumentação daquele ponto em diante do código. **`/* instr_func */`** instrui o instrumentador a instrumentar apenas os pontos de entrada e saída das funções. Por fim **`/* instr_full */`** instrui o instrumentador a inserir instruções de instrumentação em todas as funções e pontos de tomada de decisão daquele ponto em diante do código. Os comandos de instrumentação podem ser inseridos em qualquer lugar do código fonte e seu escopo é local, ou seja, eles têm validade até o final do bloco aonde foram definidos, ou então até o próximo comando de instrumentação ser encontrado. Se nenhum comando for encontrado pelo instrumentador em um determinado arquivo, então ele não insere nenhuma instrução de instrumentação estrutural naquele arquivo, ou seja, é como se **`/* instr_none */`** fosse o padrão.

Além dos três comandos de instrumentação estrutural apresentados, um quarto comando é reconhecido pelo instrumentador: o **`/* instr_user */`**. Este comando, chamado de instrumentação casual, pode ser inserido pelo usuário em qualquer lugar do código onde seja conveniente fazer uma sinalização para o sistema de monitoração. É o equivalente a escrever no código **`printf("Ponto X executado!\n");`**, só que de uma forma muito menos intrusiva. Cada vez que um **`/* instr_user */`** é encontrado no programa do usuário pelo instrumentador, um identificador de evento único é alocado para o evento e uma chamada a **`{instr(event_id);}`** é inserida no local.

Para a instrumentação contextual, a variável a ser monitorada é selecionada pelo seu identificador, que deve terminar com a cadeia de caracteres **`_TRACED`**. Assim, se uma variável chamada **`device_state`** precisa ser monitorada, então o programador deve alterar o identificador da variável para **`device_state_TRACED`**. Desta forma, quando o instrumentador encontrar a declaração desta variável, ele reconhecerá que é uma variável que precisa ser marcada, associará a ela um identificador de evento único, localizará todas as atribuições explícitas feitas a ela no código fonte, acrescentando após cada atribuição a chamada a uma das duas funções de monitoração de valores de variáveis, **`{dinstr16(data16_id,val16);}`** ou **`{dinstr32(data32_id,val32);}`**.

Para efetuar a instrumentação especulativa, o instrumentador precisa identificar as chamadas feitas às funções de biblioteca que se deseja monitorar e inserir instruções de instrumentação antes e depois da chamada. Para realizar a instrumentação especulativa o instrumentador depende de o usuário fornecer um arquivo texto contendo o nome das funções de biblioteca a monitorar.

5.3.4 FORMATOS DE INSTRUMENTAÇÃO

Os formatos de instrumentação descrevem a interface lógica entre o SUT e o monitor. Uma vez que a interface física possui 16 bits de dados, até 65.356 valores diferentes podem ser enviados. Porém, como nem todas as informações de instrumentação podem ser representadas com estes valores, a escala disponível foi dividida logicamente em três formatos de instrumentação: simples, dados e sistema operacional.

Eventos simples utilizam valores de 16 bits, de 0x0000 até 0xD9FF, ou seja, permitem a representação de até 55.808 eventos. Eventos deste formato são utilizados para as instrumentações estrutural e especulativa, sendo que o instrumentador automático de código é responsável pela atribuição de um valor único a cada evento estrutural ou especulativo do programa.

Eventos de dados utilizam valores entre 0xE000 e 0xEFFF, e são utilizados na instrumentação contextual. Neste tipo de evento, além do identificador, é necessário também enviar para a porta de medição o novo valor da variável marcada. Por este motivo mais de uma escrita na porta de medição é necessária, e os valores de eventos reservados para dados representam apenas a primeira escrita na porta de monitoração. A Tabela 4 a seguir mostra como identificar uma escrita na porta de monitoração para uma instrumentação de dado. O campo *sz* indica o tamanho da variável que está sendo marcada, sendo que um valor '0' indica uma variável de 16 bits e um valor '1' indica uma variável de 32 bits. No primeiro caso a próxima escrita na porta de monitoração será o valor da variável e, no segundo, as duas próximas escritas. A variável sendo marcada é identificada pelo campo *veid* de 11 bits, o que possibilita a monitoração de até 2.048 variáveis de 16 bits, e mais 2.048 variáveis de 32 bits. Devido ao modelo de interface física adotado, variáveis de 8 bits são tratadas como variáveis de 16 bits.

Tabela 4: Formato de eventos de dados.

Número dos bits															
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	*sz*	*veid*										

Eventos do sistema operacional utilizam valores entre 0xF000 e 0xFFFF, e foram concebidos para atender as necessidades de instrumentação de diferentes núcleos de tempo real. O formato da primeira escrita de um evento de sistema operacional na porta de medição é mostrado na Tabela 5. Com este formato é possível utilizar até 128 bits para representar um evento de sistema operacional. O campo *size* indica quantas escritas adicionais a primeira serão necessárias para descrever o evento. O campo *oseid* contém o identificador do serviço ou do grupo de serviços em questão. O significado destes eventos depende do sistema operacional ou do núcleo de tempo real envolvido.

Tabela 5: Formato de eventos de sistema operacional.

Número dos bits															
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	*size*			*oseid*								

Valores de eventos entre 0xDA00 e 0xDFFF são reservados para usos futuros e, portanto, não devem ser utilizados pelo instrumentador.

Tanto no caso de eventos de dados como no caso de eventos do sistema operacional, o evento é representado por mais de uma escrita na porta de medição. Nestes casos as escritas na porta de medição devem ser feitas em ordem e de forma atômica.

Um resumo dos valores de eventos referentes aos formatos de instrumentação é apresentado na Tabela 6.

Tabela 6: Resumo dos formatos de instrumentação.

Faixa de Valores	Significado
`0x0000 – 0xD9FF`	Eventos simples
`0xDA00 – 0xDFFF`	Eventos reservados
`0xE000 – 0xEFFF`	Eventos de dados
`0xF000 – 0xFFFF`	Eventos do sistema operacional

5.3.5 TEMPORIZAÇÃO DE EVENTOS

É tarefa do monitor híbrido marcar o instante de tempo em que um determinado evento ocorreu. Não obstante, não existe relação absoluta de tempo entre o SUT e o monitor, fazendo com que a melhor forma de temporização dos eventos seja a relativa. Um instante de tempo para o sistema de monitoração do Dyretiva é um número inteiro e não sinalizado de 32 bits, cujo significado é o número de contagens de relógio decorridas desde o evento imediatamente anterior. O relógio interno do sistema de monitoração deve funcionar a uma freqüência de 50MHz, implicando o valor de 20ns para cada contagem de relógio. Assim, o tempo máximo entre dois eventos sem estouro do contador é de aproximadamente 1,4 minutos (2^{32}x20ns). No caso de dois eventos serem espaçados por um intervalo maior do que este, então o evento especial 0xDE00 – que é um valor de evento reservado para o instrumentador – é colocado no arquivo de rastreamento pelo monitor para indicar o estouro do contador. Deste modo, o analisador do arquivo de rastreamento é capaz de calcular qualquer intervalo entre dois eventos consecutivos.

5.3.6 PROCESSO DE INSTRUMENTAÇÃO

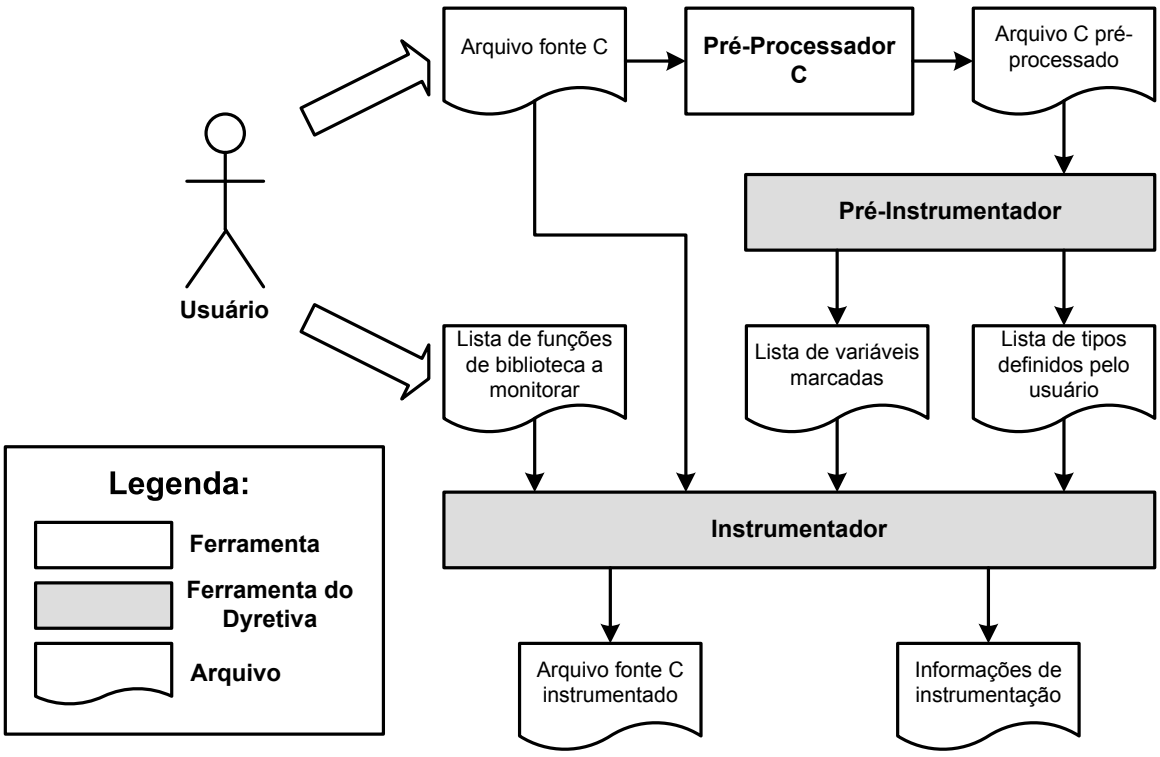


Figura 16: Fluxograma do processo de instrumentação do Dyretiva.

Em virtude das características exigidas do instrumentador automático de código fonte do usuário, um processo de instrumentação com várias etapas foi definido. Um fluxograma do processo de instrumentação adotado pelo Dyretiva é mostrado na Figura 16.

O processo de instrumentação inicia-se com o usuário providenciando o código fonte a ser instrumentado e a lista de funções de biblioteca a serem monitoradas. A partir daí, um pré-processador C padrão, como o GNU CPP, por exemplo, é utilizado no arquivo do usuário. O objetivo desta etapa é resolver algumas questões relacionadas com as especificidades da linguagem C, tais como realizar a inclusão de arquivos de cabeçalho e a substituição dos tipos definidos pelo usuário pelos tipos pré-definidos da linguagem C. O arquivo pré-processado é então apresentado para uma ferramenta do Dyretiva chamada pré-instrumentador, cujo objetivo é extrair dele a lista das variáveis marcadas e a lista dos tipos definidos pelo usuário. Quando estas duas listas ficam disponíveis, as informações nelas contidas, juntamente com o arquivo fonte original e com a lista de funções de biblioteca a serem monitoradas, são apresentadas ao instrumentador. O papel do instrumentador é modificar o código fonte do usuário, de modo a inserir as instruções de instrumentação nos locais apropriados, e também gerar um arquivo de descrição da instrumentação inserida. Este último será utilizado posteriormente pelas ferramentas de filtragem e análise do Dyretiva (ver Figura 14). Ao instrumentar o código do usuário, o instrumentador não deve modificar número ou a localização de uma linha, mantendo a relação entre as linhas do arquivo original do usuário e do arquivo instrumentado.

5.4 MODELO DE FALTA

Um modelo de falta identifica componentes do SUT e relações entre os mesmos onde se presume haver maior probabilidade de encontrar faltas, e pode basear-se em senso comum, experiência, suspeita ou análise (Seção 2.2.1). O modelo de falta do Dyretiva baseia-se em experiência e reflete três problemas comuns encontrados em ERTS: concorrência, processamento e perda de prazo.

Uma vez que as faltas esperadas no sistema decorrem de problemas causados pela concorrência e por violações da especificação temporal, uma premissa do modelo de falta do Dyretiva é tornar possível a recuperação das linhas de execução das tarefas e os tempos associados, o que pode ser obtido da monitoração do sistema operacional. Durante a análise do arquivo de rastreamento, algoritmos simulam a lógica do escalonador para verificar quando uma tarefa é selecionada para execução, que serviços do sistema operacional são

chamados por ela, e o momento em que deixa de executar. A recuperação das linhas de execução, em última instância, é a ferramenta que dá a visibilidade do comportamento dinâmico do sistema, e justifica a necessidade de conhecer em detalhes o funcionamento do sistema operacional (Seção 4.3). A tarefa de recuperação das linhas de execução baseia-se nas transições de estado das tarefas no sistema operacional. Um exemplo deste é ilustrado na Figura 52 do Anexo I, na forma de Statecharts, para o núcleo de tempo real PET#. Outra questão importante relacionada ao algoritmo de recuperação das linhas de execução é a necessidade de remontar a figura do sistema mesmo quando o arquivo de rastreamento não inicia no tempo zero do sistema, pois este é o caso comum nas situações reais de utilização.

5.4.1 FALTAS DE CONCORRÊNCIA

As faltas de concorrência podem ser causadas por três motivos: preempção, interrupções ou sincronização.

As faltas causadas por preempção ocorrem quando uma tarefa não atende seu prazo porque tarefas de maior prioridade solicitam o processador e a deixam esperando por um tempo demasiadamente longo. A aplicação do método Dyretiva auxilia na localização destas faltas identificando as preempções sofridas por cada tarefa, bem como a localização dos eventos correspondentes no arquivo de rastreamento. Com isto, o usuário pode utilizar ferramentas de visualização para identificar o motivo que levou as tarefas de maior prioridade a precisaram do processador. Uma possível solução para este tipo de falta pode ser a revisão das prioridades atribuídas para as tarefas. Outra solução pode ser o aumento do desempenho do processador.

As faltas causadas por interrupções ocorrem quando uma tarefa não atende seu prazo porque exceções ou interrupções de *hardware* solicitam o processador com muita freqüência, forçando a tarefa a ficar interrompida por um tempo demasiadamente longo. A aplicação do método Dyretiva auxilia na localização destas faltas identificando a quantidade, a posição no arquivo de rastreamento e o tempo de processamento das rotinas de atendimento. A informação temporal destas rotinas é consolidada por três valores: o tempo de execução observado no melhor caso (OBCET), o tempo de execução observado na média (OTCET) e o tempo de execução observado no pior caso (OWCET). Com estes valores, o usuário pode identificar discrepâncias entre os tempos de execução esperados, ou mesmo entre o tempo de execução no caso médio e tempo de execução no pior caso, tornando possível localizar situações ou algoritmos no código fonte que estejam executando em rotinas de atendimento e

consumindo muito tempo. Se esta for a causa do problema, é possível resolvê-lo otimizando algoritmos ou passando parte da funcionalidade das rotinas de atendimento para as tarefas. Por outro lado, se a origem do problema for uma intensa interação com o meio, é necessário reavaliar a política de atendimento de algumas interrupções. Se nenhuma destas alternativas puder ser levada a efeito, então será necessário aumentar o desempenho do processador.

As faltas causadas por sincronização ocorrem quando uma tarefa não consegue concluir o processamento de uma entrada dentro do prazo estipulado porque, de acordo com o modelo de comunicação e sincronização adotado, fica esperando em filas do sistema operacional, semáforos ou outros elementos de exclusão mútua. A aplicação do Dyretiva auxilia na localização destas faltas identificando as tarefas envolvidas em operações de comunicação ou sincronização, que podem ser feitas através de trocas de mensagens ou de memória compartilhada. No caso de troca de mensagens, as razões para atrasos podem ser um servidor de recursos sobrecarregado ou a perda de mensagens. No caso de haver um servidor de recursos sobrecarregado, é necessário rever a política de acesso dos clientes e, se necessário, multiplicar os recursos disponibilizados. No caso de haver perda de mensagens, é necessário rever o tamanho das filas, ou então rever as prioridades de tarefas envolvidas para certificar-se de que todas elas terão oportunidade de acessar a fila antes de tornar-se necessário descartar mensagens. No outro caso, que refere-se a comunicação e sincronização por memória compartilhada, o procedimento é identificar as tarefas que estão contendendo por um recurso, e definir se houve um *deadlock*, uma inanição ou uma atribuição equivocada de prioridade a uma das tarefas envolvidas.

5.4.2 FALTAS DE PROCESSAMENTO

As faltas de processamento ocorrem porque um algoritmo ou uma tarefa estão utilizando o processador por mais tempo do que o esperado. A aplicação do método Dyretiva auxilia na localização deste tipo de problema identificando os tempos gastos por funções e tarefas durante a monitoração.

As faltas de processamento em tarefas podem ser detectadas pela observação do tempo de execução das tarefas. Este tempo é um resultado da recuperação das linhas de execução e pode ser apresentado ao usuário na forma de tabelas, ou então em mais alto nível na forma de histogramas e de gráficos de setores. Um possível motivo para uma falta de processamento de tarefa pode ser o uso da espera ocupada (*busy wait*) na operação com um periférico ou com outra tarefa que demora muito para responder em determinada

circunstância, o que pode ser resolvido com uma revisão no funcionamento da tarefa faltante. Outro possível motivo para uma falta de processamento de tarefa é a atribuição de muita funcionalidade para uma mesma tarefa, fazendo com que ela gaste muito tempo no tratamento dos seus diversos canais de entrada. Uma possível solução para este problema é a divisão da tarefa que está sobrecarregada em mais tarefas.

O tempo de execução de funções, assim como o tempo utilizado na execução de interrupções, é consolidado por três valores: o OBCET, o OTCET e o OWCET. Com estes valores o usuário pode identificar quando uma função está utilizando muito tempo de processamento, bem como a possibilidade de haver um OWCET muito maior do que um OTCET para uma determinada função. Para funções que utilizam mais tempo de processamento do que o esperado, uma forma possível de solucionar o problema é melhorar o algoritmo utilizado. Para ajudar a análise do problema, informações úteis como o número de execuções de cada função e a localização de cada instância no arquivo de rastreamento devem ser fornecidas. Se a melhora do algoritmo não for possível, duas soluções alternativas podem ser estudadas: a transferência de parte da funcionalidade da função para um componente de *hardware* ou a melhora do desempenho do processador.

5.4.3 FALTAS DE PRAZO

As duas situações anteriores representam faltas comuns em ERTS, mas sua análise no modelo do Dyretiva depende basicamente do usuário. As faltas de prazo, ao contrário, procuram representar a especificação temporal do sistema, transferindo para as ferramentas do método a responsabilidade pela procura de faltas temporais.

Para representar a especificação temporal do sistema, o Dyretiva descreve uma restrição temporal por cinco atributos. O primeiro é o evento que marca o começo da restrição temporal, chamado de evento inicial ($\mathbf{E_i}$). O segundo atributo é o evento que caracteriza o encerramento da restrição temporal, chamado de evento final ($\mathbf{E_f}$). O terceiro atributo é o conjunto das tarefas relacionadas com a restrição temporal, chamado de conjunto das tarefas envolvidas ($\mathbf{\Phi}$). Este atributo parte do princípio que uma restrição temporal não precisa envolver apenas uma única tarefa. O quarto atributo é o prazo ($\mathbf{t_d}$), definido como o maior tempo entre a ocorrência do evento inicial até a ocorrência do evento final, e que ainda atende aos requisitos temporais. O quinto e último atributo é o período ($\mathbf{t_T}$), definido como o intervalo esperado entre dois eventos iniciais, caso a restrição temporal seja periódica. Neste modelo, o prazo e o período não precisam ser necessariamente iguais. Também é permitido

que uma restrição temporal seja aperiódica, o que pode ser representado definindo $\mathbf{t_T}$ com valor igual a zero.

Para exemplificar, tome-se como referência uma restrição temporal com a seguinte especificação: $\mathbf{E_i} = \mathbf{e_1}$, $\mathbf{E_f} = \mathbf{e_2}$, $\mathbf{\Phi}$ = { T1, T4, T5 }, $\mathbf{t_d}$ = 12 unidades de tempo e $\mathbf{t_T}$ = 14 unidades de tempo. Uma das possíveis seqüências de execução desta restrição temporal é mostrada na Figura 17.

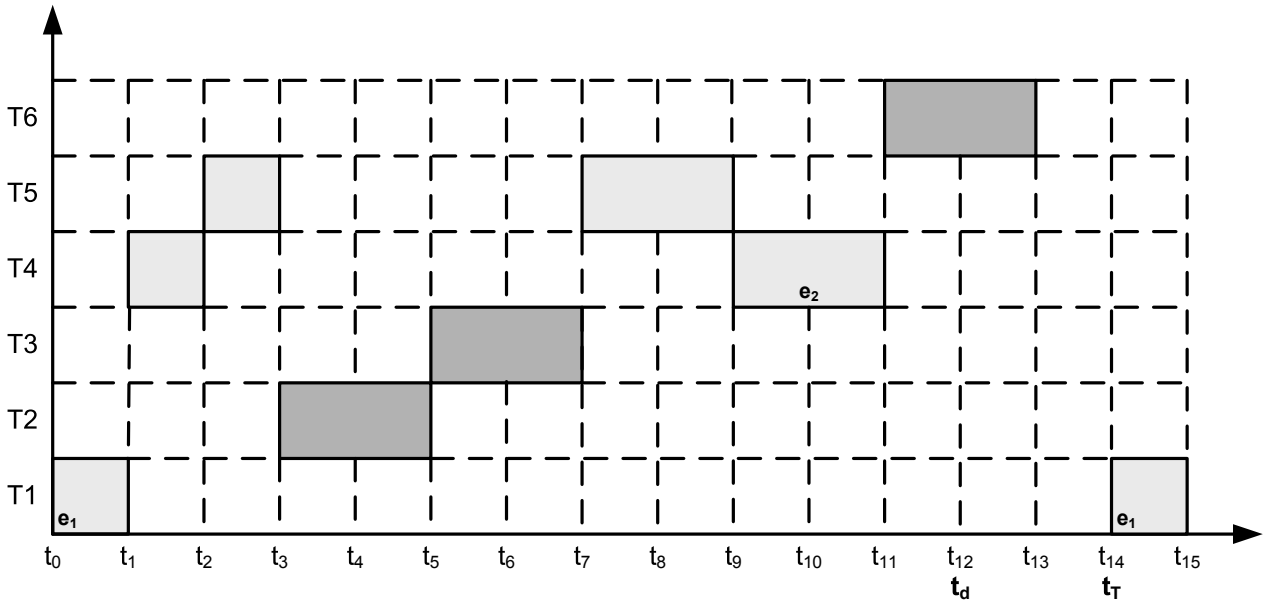


Figura 17: Seqüência de execução da restrição temporal.

Na Figura 17 as tarefas mostradas em cinza claro são as tarefas envolvidas, enquanto as tarefas mostradas em cinza escuro não possuem relação com a restrição temporal ilustrada. O eixo vertical mostra as tarefas em ordem de prioridade, sendo T1 a tarefa mais prioritária e T6 a tarefa menos prioritária. As unidades de tempo são representadas no eixo horizontal na forma t_n, com n variando de 0 até 15, totalizando 15 unidades de tempo.

A Figura 17 mostra uma situação em que a restrição temporal cumpre o prazo, pois $\mathbf{e_2}$ ocorre antes de $\mathbf{t_d}$. Além disso, outras informações importantes podem ser inferidas com base no modelo. A primeira é a obtenção do tempo de computação ($\mathbf{t_c}$) da restrição temporal, que é obtido somando-se todos os intervalos entre $\mathbf{e_1}$ e $\mathbf{e_2}$ em que uma das tarefas envolvidas esteja em execução, ou seja, $\mathbf{t_c} = [(t_1-t_0) + (t_2-t_1) + (t_3-t_2) + (t_9-t_7) + (t_{10}-t_9)] = 6$ unidades de tempo. Outra informação importante que pode ser extraída do modelo é o tempo de interferência ($\mathbf{t_i}$) da restrição temporal, ou seja, o tempo após o evento inicial em que a restrição não estava em execução porque existiam tarefas de mais alta prioridade utilizando o

processador. No exemplo mostrado, $\mathbf{t_i} = [(t_7\text{-}t_3)] = 4$ unidades de tempo. Finalmente, uma última informação obtida do modelo é o desvio no período da restrição temporal (*jitter*), que no exemplo mostrado é zero em virtude de $\mathbf{e_1}$ ocorrer pela segunda vez exatamente $\mathbf{t_T}$ unidades de tempo após a primeira ocorrência.

Com base neste modelo de faltas de prazo, uma restrição temporal não é atendida quando o prazo $\mathbf{t_d}$ acontece fora do intervalo designado. Isto pode ter ocorrido porque $\mathbf{t_c}$ foi muito grande, implicando uma falta de processamento, ou então porque $\mathbf{t_i}$ foi muito grande, implicando uma falta de concorrência. Além disso, é possível também inferir o desvio de período (*jitter*) da restrição temporal, caso esta seja periódica.

5.5 PASSOS DO MÉTODO

De acordo com o que foi apresentado neste capítulo, a aplicação do método Dyretiva pode ser descrita através dos passos a seguir, identificados pelos algarismos **i** a **xii**.

- **i.** Prever no *hardware* do sistema embarcado uma porta de monitoração (Seção 5.3.1).
- **ii.** Gerar uma versão instrumentada do núcleo de tempo real ou do sistema operacional utilizado, identificando os pontos de chaveamentos entre tarefas, preempções, interrupções e acessos a regiões críticas (Seções 2.2.2 e 5.3.4).
- **iii.** Elaborar o arquivo de cabeçalho **`"instr.h"`**, cujo conteúdo depende da plataforma de *hardware*. Este arquivo deve conter as macros ou os protótipos das funções **`instr`**, **`dinstr16`** e **`dinstr32`** (Seção 5.3.2). Chamadas a estas macros ou funções serão inseridas no código posteriormente pelo instrumentador. A implementação típica destas funções é uma ou mais escritas de dados consecutivas em um mesmo endereço de memória, que é o endereço físico de onde se encontra a porta de monitoração na plataforma de *hardware*.
- **iv.** Identificar as partes do código da aplicação que precisam ser instrumentadas e o nível de instrumentação requerido em cada parte, incluindo os comandos de instrumentação adequados nas diferentes partes do código (Seção 5.3.3).
- **v.** Instrumentar o código fonte da aplicação, conforme definido pelo processo de instrumentação (Seção 5.3.6).
- **vi.** Descrever as restrições temporais do sistema da forma requerida pelo modelo de falta do Dyretiva (Seção 5.4.3).

vii. Montar um ambiente de execução que seja composto pela plataforma de *hardware* do sistema embarcado conectada ao monitor híbrido através da porta de medição (Figura 14).

viii. Selecionar um dos casos de teste para execução.

ix. Executar o caso de teste selecionado, utilizando as versões instrumentadas da aplicação e do sistema operacional, capturar os eventos gerados com o monitor híbrido.

x. Ler do monitor híbrido o arquivo de rastreamento gerado pela execução do caso de teste. Este arquivo é formado pelos eventos capturados e pelo instante de tempo em que cada evento ocorreu. A temporização dos eventos é feita pelo monitor híbrido conforme descrito na Seção 5.3.5.

xi. Filtrar e analisar as informações do arquivo de rastreamento, utilizando para isto as informações geradas pelo instrumentador de código e a descrição das restrições temporais. Os resultados deste processo de filtragem e análise são as informações sobre o comportamento dinâmico do sistema e a lista das eventuais violações de restrições temporais observadas durante a execução do caso de teste.

xii. Apresentar os resultados obtidos na análise do arquivo de rastreamento ao usuário, permitindo que ele visualize o comportamento dinâmico do sistema e identifique as eventuais faltas de concorrência ou de prazo. Uma vez que faltas tenham sido identificadas, elas devem ser registradas e para posterior correção pela equipe de desenvolvimento. Se nenhuma falta for identificada, então o próximo caso de teste deve ser escolhido, voltando-se ao passo **viii**. Esta seqüência é repetida até que todos os casos de teste tenham sido executados sem que tenham sido observadas faltas temporais ou lógicas.

Seguindo estes passos é possível garantir que o produto atende a todos os requisitos temporais exercitados pelos casos de teste previstos no plano de teste. Além disso, a abordagem de monitoração do Dyretiva permite minimizar os recursos utilizados para teste e a intrusão causada no sistema monitorado, tornando possível manter no produto as instruções de instrumentação do sistema operacional e da aplicação, eliminando o efeito sonda.

5.6 RESUMO

Neste capítulo foi apresentado o Dyretiva, que é um método elaborado para ser utilizado durante as fases de testes e verificação das restrições temporais em sistemas embarcados que operam em tempo real. Inicialmente foram definidos os requisitos do método, utilizando como premissas as necessidades de testes e as características particulares dos ERTS. Definidos os requisitos, foi desenhado o modelo de utilização, que tomou como base o processo de desenvolvimento do LIT descrito na Seção 2.1. A seguir foram apresentados os dois elementos principais do Dyretiva: a abordagem de monitoração e o modelo de falta. Finalmente, com base no modelo de utilização, na abordagem de monitoração e no modelo de falta, foram apresentados os passos a serem seguidos para aplicar o método.

A abordagem de monitoração do Dyretiva define e padroniza o processo de instrumentação e de coleta de dados do SUT, de forma a torná-lo adequado aos sistemas embarcados e, ao mesmo tempo, prover a flexibilidade necessária para que ele possa ser aplicado em sistemas com diferentes arquiteturas de processadores e RTOS. Buscando uma minimização da intrusão causada, o Dyretiva prevê que as instruções de instrumentação inseridas no código do usuário e no sistema operacional não sejam removidas do SUT, fazendo com que o sistema testado e o sistema entregue sejam os mesmos. Esta abordagem elimina por completo a possibilidade de inserir o efeito sonda no sistema observado.

O modelo de falta identifica as relações e os componentes do ERTS onde se presume haver maior probabilidade de encontrar faltas. As faltas procuradas pelo Dyretiva são faltas de concorrência, faltas de processamento e faltas de prazo.

Comparando o Dyretiva com trabalhos relacionados apresentados na Seção 4.4, pode-se incluir o Dyretiva na comparação feita na Tabela 3, obtendo-se com isto a Tabela 7.

Tabela 7: Comparação dos métodos e ferramentas estudados com o Dyretiva.

Método e/ou Ferramenta		Tipo de Monitoração	Independência da arquitetura do Processador	Suporte a diferentes RTOS	Independência do ambiente de desenvolvimento	Instrumen-tação Seletiva	Intrusão	Testes Estrutu-rais
TLM		Híbrida	Não	Não	Sim	Sim	Baixa	Sim
VDS		Hardware	Não	Não	Sim	Não	Nula	Não
Code TEST	HWIC	Híbrida	Não	Sim	Não	Não	Baixa	Sim
	SWIC	Software	Sim	Não	Não	Não	Alta	Sim
WindView		Software	Sim	Não	Não	Não	Alta	Não
Reprise		Software	Sim	Sim	Sim	Não	Alta	Não
Dyretiva		Híbrida	Sim	Sim	Sim	Sim	Baixa	Sim

Uma vez definidos e descritos os elementos do Dyretiva, o próximo capítulo apresenta um conjunto de ferramentas construído com o objetivo de permitir a aplicação do método. Este conjunto de ferramentas é chamado SoftScope.

6 SOFTSCOPE: FERRAMENTAS DE APOIO AO DYRETIVA

Este capítulo apresenta o SoftScope, um conjunto de ferramentas implementado com o objetivo de permitir a aplicação do Dyretiva. A funcionalidade de cada ferramenta do SoftScope baseia-se em uma interpretação do método, e suas implementações foram feitas a partir de ambientes de desenvolvimento e programas que estavam disponíveis na época. Deste modo, pode-se afirmar que é possível gerar outros conjuntos de ferramentas que permitem aplicar o método Dyretiva igualmente.

O SoftScope é composto de seis ferramentas: um pré-instrumentador de código fonte C, um instrumentador de código fonte C, um monitor híbrido externo, um programa de acesso ao monitor, programas de filtragem e análise dos dados coletados, e um programa de apresentação dos resultados.

O pré-instrumentador e o instrumentador de código fonte são utilizados na instrumentação do código fonte do usuário, conforme definido pelo processo de instrumentação do Dyretiva mostrado na Figura 16. As demais ferramentas são utilizadas durante o ciclo de depuração e teste do SUT, conforme definido pelo fluxograma de utilização do Dyretiva e indicado pelas caixas cinzentas da Figura 14.

6.1 PRÉ-INSTRUMENTADOR

O objetivo do pré-instrumentador é extrair de um arquivo C, que tenha sido previamente pré-processado, duas informações importantes para o processo de instrumentação: uma lista das variáveis monitoradas e uma lista dos tipos definidos pelo usuário. Estas informações serão utilizadas posteriormente pelo instrumentador para realizar a instrumentação contextual.

O pré-instrumentador é implementado como um compilador ANSI C simplificado, capaz de entender tanto a linguagem ANSI C como a sintaxe do arquivo pré-processado. A estrutura interna do pré-instrumentador pode ser considerada clássica, contendo um analisador léxico e um analisador sintático, e sua implementação utiliza dois conhecidos programas GNU: o flex e o bison.

O trabalho do pré-instrumentador pode ser entendido melhor a partir de um exemplo. Considere o arquivo fonte C, chamado de `arquivo.c`, mostrado na Figura 18. No início de `arquivo.c`, o arquivo de cabeçalho `tipos.h` é incluído. Este arquivo de cabeçalho é composto pelas duas linhas de código mostradas na Figura 19, definindo os tipos `INT16` e `INT32`.

```
#include "tipos.h"

#define LIMITE_CONTADOR  16384

#define ESTADO_0            0
#define ESTADO_1            1
#define ESTADO_2            2

INT16 contador_TRACED = 0;
INT32 estado_TRACED = ESTADO_0;

/* instr_func */
int AtualizaContador ( void )
{
    contador_TRACED++;
    if ( contador_TRACED >= LIMITE_CONTADOR ) {
        contador_TRACED = 0;
        return (0);
    }
    FuncBib( );
    return (1);
}

/* instr_full */
void AtualizaEstado ( void )
{
    switch (estado_TRACED)
    {
        case ESTADO_0:
            estado_TRACED = ESTADO_1;
            break;
        case ESTADO_1:
            estado_TRACED = ESTADO_2;
            break;
        case ESTADO_2:
            estado_TRACED = ESTADO_0;
            break;
        default:
            estado_TRACED = ESTADO_0;
            break;
    }
}
```

Figura 18: Conteúdo do arquivo fonte C `arquivo.c`.

```
typedef signed short INT16;
typedef signed int   INT32;
```

Figura 19: Conteúdo do arquivo de cabeçalho `tipos.h`.

Seguindo o fluxograma da Figura 16, aplica-se o pré-processador C ao arquivo `arquivo.c`, obtendo-se como resultado o arquivo `arquivo.i` mostrado na Figura 20. O pré-processador utilizado neste exemplo é o GNU CPP, que não faz parte do SoftScope.

```
# 1 "arquivo.c"
# 1 "<built-in>"
# 1 "<command line>"
# 1 "arquivo.c"
# 1 "tipos.h" 1
typedef signed short INT16;
typedef signed int INT32;
# 2 "arquivo.c" 2

INT16 contador_TRACED = 0;
INT32 estado_TRACED = 0;

int AtualizaContador ( void )
{
    contador_TRACED ++;
    if ( contador_TRACED >= 16384 ) {
        contador_TRACED = 0;
        return (0);
    }
    FuncBib( );
    return (1);
}

void AtualizaEstado ( void )
{
    switch (estado_TRACED)
    {
        case 0:
            estado_TRACED = 1;
            break;
        case 1:
            estado_TRACED = 2;
            break;
        case 2:
            estado_TRACED = 0;
            break;
        default:
            estado_TRACED = 0;
            break;
    }
}
```

Figura 20: Conteúdo do arquivo pré-processado `arquivo.i`.

A Figura 20 mostra o trabalho do pré-processador C, que consiste em incluir o conteúdo de arquivos de cabeçalho, substituir macros, remover comentários e incluir informações específicas do pré-processador nas linhas que começam com o caractere `#`.

O arquivo pré-processado `arquivo.i` contém todas as informações necessárias para a instrumentação contextual. Quando ele é apresentado ao pré-instrumentador, dois arquivos de saída são gerados: `arquivo.typ` e `project.dat`. Estes arquivos contêm os tipos definidos pelo usuário e a lista das variáveis monitoradas, respectivamente. O conteúdo deles para o exemplo considerado nesta seção é mostrado nas Figura 21 e Figura 22.

```
INT16, signed short, 16;
INT32, signed int, 32;
```

Figura 21: Conteúdo do arquivo `arquivo.typ`.

```
contador_TRACED, 16, 57344;
estado_TRACED, 32, 59392;
```

Figura 22: Conteúdo do arquivo `project.dat`.

Conforme pode ser observado na Figura 21, cada linha do arquivo de tipos contém três argumentos, que informam o nome do tipo definido pelo usuário, seu tipo original em linguagem C, e sua largura em bits. Estas informações serão utilizadas posteriormente para decidir, no caso de haver uma variável monitorada com um tipo definido pelo usuário, qual função de instrumentação de dados deve ser utilizada: `dinstr16` ou `dinstr32`. O pré-instrumentador cria um arquivo de tipos para cada arquivo a ser instrumentado, uma vez que a linguagem C permite que o usuário redefina tipos localmente, fazendo com que tipos com nomes iguais possam ter tamanhos diferentes em diferentes arquivos fonte.

A Figura 22 mostra o formato da lista das variáveis monitoradas. Cada linha deste arquivo contém três argumentos: o nome da variável, o tamanho da variável e o número do evento que a identifica. No exemplo, `contador_TRACED` é uma variável de 16 bits que possui número de evento `57344` (`0xE000`), enquanto `estado_TRACED` é uma variável de 32 bits que possui número de evento `59392` (`0xE800`). O significado dos valores referentes aos eventos de dados encontra-se na Tabela 4 (Seção 5.3.4). Ao contrário do arquivo de tipos, o arquivo `project.dat` é único para todo o projeto. Isto é necessário para que o pré-instrumentador possa alocar um número único para cada evento de dados. Por outro lado, esta abordagem cria a limitação de que cada variável monitorada deve ter um nome único ao longo de todo o projeto, já que duas variáveis com o mesmo nome em diferentes locais do código seriam entendidas como sendo a mesma variável pelo pré-instrumentador.

6.2 INSTRUMENTADOR

O instrumentador tem o objetivo de inserir instruções de instrumentação estrutural, contextual e especulativa no código fonte do usuário. Ele recebe como entrada quatro arquivos: a lista de funções de biblioteca a monitorar, o arquivo fonte original do usuário, o arquivo de tipos definidos pelo usuário e o arquivo contendo a lista das variáveis monitoradas. Usando estes quatro arquivos e os comandos de instrumentação inseridos no código fonte pelo próprio usuário, o instrumentador gera duas saídas: o arquivo fonte do

usuário instrumentado e o arquivo com a descrição das instruções de instrumentação inseridas. Este fluxo é mostrado na Figura 16.

O instrumentador, assim como o pré-instrumentador, é implementado como um compilador ANSI C simplificado. Sua estrutura interna também é formada por um analisador léxico e um analisador sintático, sendo construído com os programas GNU flex e bison.

Para mostrar o uso da instrumentação especulativa, o exemplo utilizado neste capítulo inclui a chamada a uma suposta função de biblioteca, cujo nome é `FuncBib`, que pode ser encontrada na linha número 20 do arquivo `arquivo.c` mostrado na Figura 18. Para que ela seja reconhecida pelo instrumentador como uma função de biblioteca, um arquivo contendo a lista das funções de biblioteca, chamado `list.txt`, precisa ser fornecido, caso contrário o instrumentador assume que não existem funções de biblioteca a monitorar. Uma vez que o ANSI C não suporta funções sobrecarregadas, o conteúdo de `list.txt` é apenas o nome de cada função monitorada.

Ao aplicar o instrumentador ao arquivo `arquivo.c`, mostrado na Figura 18, utilizando como entradas o arquivo `list.txt` descrito no parágrafo anterior, o arquivo de tipos mostrado na Figura 21 e o arquivo das variáveis monitoradas apresentado na Figura 22, obtém-se como resultados o arquivo `arquivo.ins.c`, mostrado na Figura 23, e o arquivo `project.dsc`, mostrado na Figura 24.

O arquivo instrumentado da Figura 23 mostra a atuação do instrumentador. A instrumentação contextual é feita conforme os arquivos fornecidos pelo pré-instrumentador, com a inclusão de chamadas às funções `dinstr16` e `dinstr32` logo após qualquer atribuição a uma das variáveis monitoradas. Por outro lado, a instrumentação estrutural é feita com base nos comandos de instrumentação *`/* instr_func */`* e *`/* instr_full */`*, inseridos no programa pelo usuário. Como se pode observar, a função `AtualizaContador` é instrumentada apenas na entrada e nas possíveis saídas, enquanto a função `AtualizaEstado` é instrumentada em todos os pontos de decisão, tornando possível o rastreamento de todos os caminhos de execução. Finalmente a instrumentação especulativa implica a geração de dois eventos por função monitorada. No caso do exemplo exposto, os eventos 1 e 2 foram atribuídos à chamada e ao retorno da função `FuncBib`, respectivamente, sendo inseridos no código do usuário sempre antes e depois de cada chamada.

```
#include "tipos.h"

#define LIMITE_CONTADOR  16384

#define ESTADO_0            0
#define ESTADO_1            1
#define ESTADO_2            2

INT16 contador_TRACED;
INT32 estado_TRACED;

/* instr_func */
int AtualizaContador ( void ){ instr(0x5);
{
    contador_TRACED++; dinstr16(0xE000, contador_TRACED);
    if ( contador_TRACED >= LIMITE_CONTADOR ) {
        contador_TRACED = 0; dinstr16(0xE000, contador_TRACED); instr(0x3);
        return (0);
    } instr(0x1);
    FuncBib( ); instr(0x2); instr(0x4);
    return (1);
} }

/* instr_full */
void AtualizaEstado ( void ){ instr(0x10);
{instr(0xE);
    switch (estado_TRACED)
    {
        case ESTADO_0: instr(0x6);
            estado_TRACED = ESTADO_1; dinstr32(0xE800, estado_TRACED); instr(0x7);
            break;
        case ESTADO_1: instr(0x8);
            estado_TRACED = ESTADO_2; dinstr32(0xE800, estado_TRACED); instr(0x9);
            break;
        case ESTADO_2: instr(0xA);
            estado_TRACED = ESTADO_0; dinstr32(0xE800, estado_TRACED); instr(0xB);
            break;
        default: instr(0xC);
            estado_TRACED = ESTADO_0; dinstr32(0xE800, estado_TRACED); instr(0xD);
            break;
    } instr(0xF);
} instr(0x11); }
```

Figura 23: Conteúdo do arquivo `arquivo.ins.c`.

Para permitir o rastreamento do programa, o instrumentador também gera o arquivo de descrição da instrumentação, conforme mostrado na Figura 24. Este arquivo sempre começa com uma linha de sintaxe conhecida, que contém o número de eventos utilizados pelo instrumentador no programa do usuário. A partir daí, todas as linhas possuem quatro argumentos, separados por vírgulas e terminando com um ponto e vírgula. O primeiro argumento é o número do evento, o segundo é o tipo do evento, o terceiro é o nome da função em que o evento foi inserido, e o quarto é o número da linha do arquivo fonte do usuário aonde a instrumentação foi inserida. O número do evento é único em todo o programa, permitindo a identificação inequívoca de cada evento. Já os possíveis tipos de eventos existentes são mostrados na Tabela 8.

```
[MAX_EVENT = 17]
-1, FILE, arquivo.c, -1;
5, 1, AtualizaContador, 13;
3, 19, AtualizaContador, 18;
1, 22, AtualizaContador, 20;
2, 23, AtualizaContador, 20;
4, 19, AtualizaContador, 21;
56831, 2, AtualizaContador, 22;
16, 1, AtualizaEstado, 25;
14, 8, AtualizaEstado, 27;
6, 3, AtualizaEstado, 29;
7, 18, AtualizaEstado, 31;
8, 3, AtualizaEstado, 32;
9, 18, AtualizaEstado, 34;
10, 3, AtualizaEstado, 35;
11, 18, AtualizaEstado, 37;
12, 4, AtualizaEstado, 38;
13, 18, AtualizaEstado, 40;
15, 9, AtualizaEstado, 41;
17, 2, AtualizaEstado, 42;
```

Figura 24: Conteúdo do arquivo `project.dsc`.

Tabela 8: Tipos de eventos utilizados na instrumentação.

Tipo do Evento	Significado
1	Início da execução de uma função.
2	Final da execução de uma função.
3	Cláusula `case` executada.
4	Cláusula `default` executada.
5	Início da execução de um `if`.
6	Fim da execução de um `if`.
7	Início da execução de um `else`.
8	Início da execução de um `switch`.
9	Final da execução de um `switch`.
10	Início da execução de um laço `while`.
11	Final da execução de um laço `while`.
12	Início da execução de um laço `do`.
13	Final da execução de um laço `do`.
14	Início da execução de um laço `for`.
15	Final da execução de um laço `for`.
16	Cláusula `goto` executada.
17	Cláusula `continue` executada.
18	Cláusula `break` executada.
19	Cláusula `return` executada.
20	Instrumentação casual executada.
21	Valor reservado.
22	Início de chamada de função de biblioteca.
23	Retorno de chamada de função de biblioteca.

Duas linhas no arquivo de descrição da Figura 24 merecem destaque. A primeira é a que possui número de evento `-1`. Esta linha, ao contrário das demais, não descreve um

evento do programa, mas sim informa ao programa que lê o arquivo de descrição qual arquivo fonte do usuário está sendo descrito a partir deste ponto. Neste caso, o segundo argumento possui o valor fixo `FILE`, o terceiro argumento é o nome do arquivo que vai ser descrito, e o quarto argumento também possui o valor fixo `-1`. Assim como o arquivo que contém a lista das variáveis monitoradas, o arquivo de descrição também é único para um determinado conjunto de arquivos que compõe um projeto.

Outra linha que merece destaque é a que começa com o valor `56831` (`0xDDFF`). Esta linha também não se refere a nenhum evento do programa, mas sim a um valor reservado (Seção 5.3.4) cujo significado é o fim de uma função monitorada terminada com a instrução `return` antes do fechamento de chaves. Se esta abordagem não fosse adotada e instrumentação fosse inserida neste ponto, seria gerada uma advertência na compilação do arquivo instrumentado, pois o compilador indicaria que tal linha de código nunca seria executada, e ainda haveria desperdício de um número de evento. Por esta razão, toda vez que uma função termina com um `return` o instumentador indica o final da função com o evento `0xDDFF` no arquivo de descrição e não insere nenhum tipo de instrumentação no local correspondente.

6.3 MONITOR HÍBRIDO

O SoftScope utiliza um monitor híbrido externo, chamado MIMO (*Minimal Invasive Monitor*), para realizar a captura das instruções de instrumentação enviadas pelo SUT para a porta de monitoração. Os requisitos do MIMO são:

- Tratar solicitações do usuário, vindas através do programa de controle do monitor (Figura 14, Seção 5.2).
- Capturar eventos vindos da porta de medição (Seção 5.3.1) do SUT.
- Temporizar os eventos capturados a partir de um contador interno de largura 32-bits e período 20ns (Seção 5.3.5).
- Armazenar os eventos capturados em uma memória local até que o usuário os transfira para sua estação de trabalho.

Uma vez que o MIMO possui requisitos de *hardware* específicos, ele foi implementado com lógica programável, utilizando os conceitos de SoC (*System on Chip*). Um diagrama em blocos do MIMO é mostrado na Figura 25.

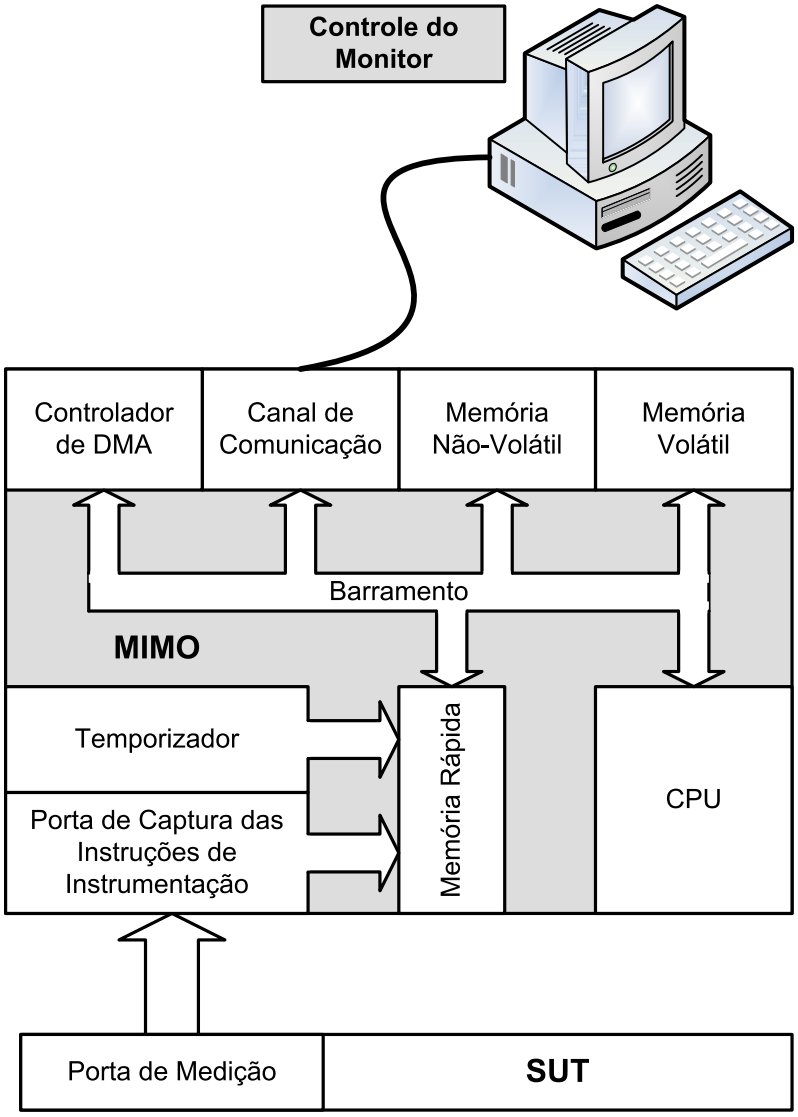


Figura 25: Diagrama em blocos do MIMO.

O MIMO possui duas interfaces externas: uma com o programa de controle do monitor, que executa em uma estação de trabalho e é controlado pelo usuário, e outra com a porta de medição do SUT. Todos os elementos internos do MIMO, conforme mostrados na Figura 25, são implementados em uma FPGA (*Field Programmable Gate Array*), com exceção das memórias volátil e não-volátil.

O MIMO é controlado por um processador que, assim como a maioria dos outros componentes, é implementado dentro da FPGA. A função do processador é coordenar o funcionamento do monitor, programar os diversos periféricos envolvidos nas transferências internas de dados e tratar o canal de comunicação com o programa de controle de monitor. Considerando esta característica, o MIMO pode ser visto como um sistema embarcado completo, contando com seu próprio processador, memórias, canais de comunicação, periféricos integrados e *software* embarcado.

A interface física entre o SUT e o MIMO é feita com um cabo plano cujos sinais de *hardware* são descritos na Seção 5.3.1. Quando em funcionamento, o MIMO decodifica os eventos enviados pelo SUT para a porta de medição, de acordo com o formato descrito na Seção 5.3.4, e anota o instante de tempo em que ocorreram, conforme descrito na Seção 5.3.5. Estas duas informações são armazenadas em uma memória de acesso rápido, evitando que o elemento amostrador da porta de medição fique indisponível por um tempo maior do que a intervalo entre dois eventos consecutivos. Quando um número de eventos suficiente é armazenado na memória rápida, o processador programa o controlador de DMA para transferi-los da memória rápida para a memória principal. A memória rápida possui duas portas de acesso: uma pelo lado do amostrador de eventos e outra pelo barramento onde está o controlador de DMA. Deste modo, a amostragem não precisa parar enquanto o DMA está em funcionamento. Além disso, a memória rápida é dividida em partes, permitindo que enquanto um DMA é feito de uma das partes para a memória principal, o amostrador continue depositando eventos em outra parte.

A implementação do *hardware* do MIMO, assim como uma análise de seu desempenho, envolveram também as áreas de lógica programável e SoC no CPGEI, além da área de sistemas embarcados. Uma publicação sobre aspectos de *hardware* do MIMO pode ser encontrada em [Copetti 07].

A outra interface externa do MIMO é com o programa de controle do monitor, que é comandado pelo usuário. Nesta interface o programa de controle do monitor é o mestre da comunicação e o monitor é o escravo. A interface física entre ambos na primeira versão do SoftScope é uma porta serial padrão RS-232, sobre a qual são trocadas mensagens entre mestre e escravo. A interface lógica entre o MIMO e o programa de controle do monitor é composta por seis comandos básicos: ecoar, ler configuração, iniciar amostragem, finalizar amostragem, ler eventos e reiniciar.

Um comando de ecoar é enviado pelo programa de controle, pedindo que o monitor simplesmente responda com a mesma mensagem que foi enviada. Este comando serve para verificar se o monitor está conectado e pronto para operar. Um comando de ler configuração solicita que o monitor envie sua configuração e estado atuais, incluindo o estado do monitor (parado ou coletando eventos), o tamanho da memória disponível para o armazenamento de eventos, a quantidade de memória de armazenamento já preenchida com eventos e o número de eventos capturados. Um comando de iniciar amostragem solicita que o monitor comece a armazenar na memória os eventos detectados na porta de medição. Um comando de finalizar amostragem solicita que o monitor pare de armazenar na memória os eventos detectados na porta de medição. Um comando de ler eventos solicita que o monitor leia e envie para o programa de controle um determinado trecho da memória de eventos. Finalmente, um comando de reiniciar solicita que o monitor tome as ações necessárias para retornar ao seu estado inicial, incluindo periféricos, variáveis de controle, e máquinas de estado. O comportamento esperado do MIMO pode ser representado pelo Statechart [Harel 87] mostrado na Figura 26.

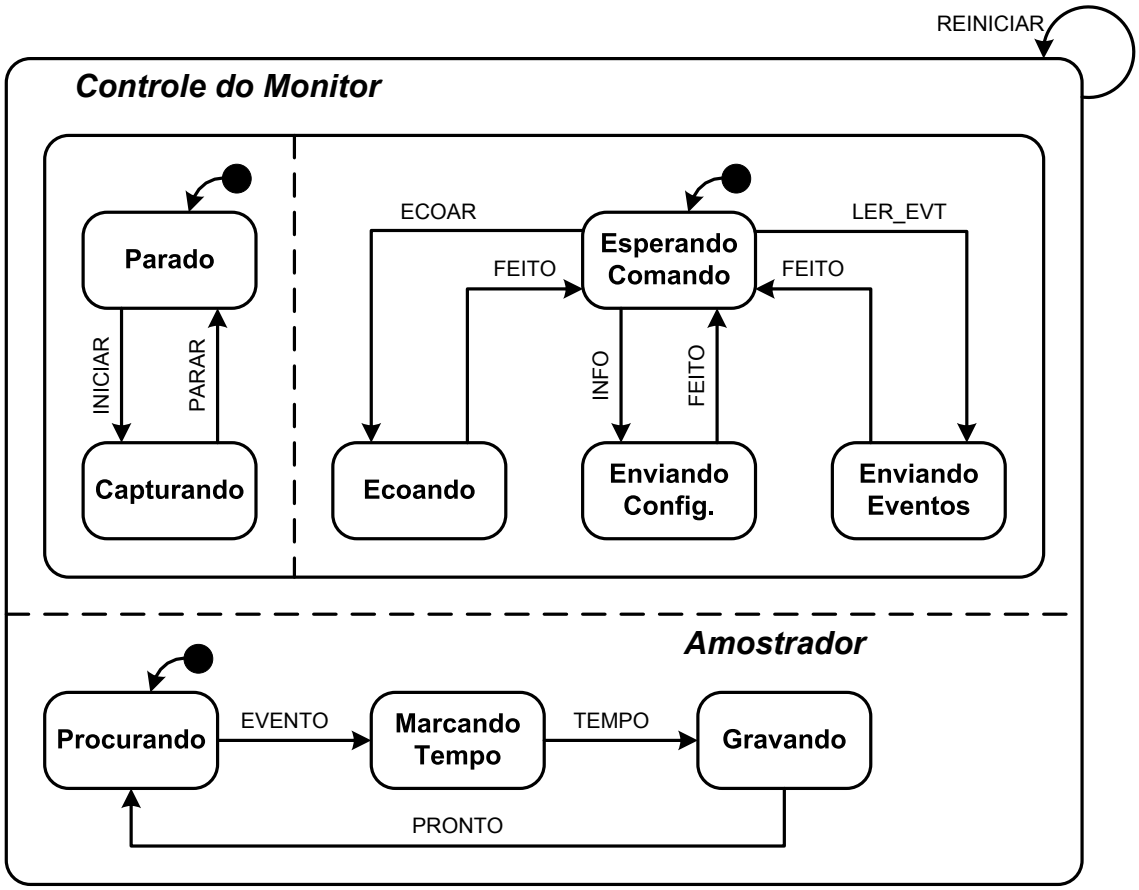


Figura 26: Diagrama de estados do MIMO.

A Figura 26 mostra que o MIMO possui duas atividades independentes: o controle do monitor e o amostrador. O controle do monitor é uma atividade executada pelo processador do MIMO, de acordo com comandos vindos do usuário através do programa de controle do monitor. Já o amostrador é uma atividade implementada com um componente dedicado de *hardware*, cujo objetivo é detectar, temporizar e armazenar eventos vindos do SUT. Complementando o diagrama de estados do MIMO, a Tabela 9 descreve o significado de cada evento mostrado na Figura 26.

Tabela 9: Descrição dos eventos de controle do MIMO.

Evento	Descrição
REINICIAR	Comando para reiniciar o monitor.
EVENTO	Indicação de que um novo evento foi detectado na porta de medição.
TEMPO	Indicação de que o relógio de temporização de eventos foi lido e seu valor está disponível.
PRONTO	Indicação de que um evento e seu instante de tempo foram adequadamente tratados pela máquina de estados de amostragem, e que esta está livre para começar a procurar pelo próximo evento.
INICIAR	Comando de iniciar amostragem.
PARAR	Comando de parar amostragem. Este evento pode ser gerado a partir de uma solicitação do usuário, ou pelo próprio monitor quando a memória de eventos fica lotada.
ECOAR	Comando para ecoar a mensagem recebida.
INFO	Comando de leitura das informações de configuração.
LER_EVT	Comando de leitura da memória de eventos.
FEITO	Indicação de que um comando de ECOAR, INFO, ou LER_EVT foi respondido para o programa de controle do monitor.

Outras informações sobre o protocolo de comunicação entre o programa de controle do monitor e o MIMO, bem como a descrição das mensagens trocadas entre ambos, podem ser encontradas no Anexo II deste trabalho.

6.4 MONITOR BASEADO EM SOFTWARE

Em algumas situações não é possível utilizar a monitoração híbrida em virtude do módulo microprocessado ser legado de projetos anteriores, ou utilizar *hardware* de prateleira. Na grande maioria destes casos, os benefícios da monitoração híbrida não podem ser utilizados porque o SUT não implementa a porta de medição. Para sobrepujar estas limitações e utilizar o Dyretiva, ainda que apenas em parte do processo de teste e verificação

do sistema embarcado, um monitor baseado em *software* é previsto como parte do SoftScope.

O principal requisito do monitor baseado em *software* no SoftScope é que as interfaces externa sejam preservadas, ou seja, que nem o programa de controle do monitor e nem o processo de instrumentação da aplicação do usuário sejam modificados para funcionar adequadamente com o monitor baseado em *software*. Para que isto ocorra, o SUT deve disponibilizar os seguintes recursos:

- Tempo de processamento para coletar, temporizar e armazenar eventos.
- Espaço de memória para o armazenamento de eventos.
- Um canal de comunicação para trocar informações com o programa de controle do monitor
- Uma implementação da máquina de estados de controle do monitor.

Do ponto de vista da aplicação sendo monitorada, o monitor baseado em *software* deve implementar as funções `instr`, `dinstr16` e `dinstr32` (Seção 5.3.2). O trabalho destas funções é anotar o evento ocorrido, ler um relógio interno do sistema, adequar de relógio lido para torná-lo compatível com a base de tempo de 20ns utilizada pelo MIMO, e armazenar evento e instante de tempo na memória reservada para esta finalidade.

Já do ponto de vista do programa de controle do monitor, o SUT deve disponibilizar um canal de comunicação RS-232 que utilize o protocolo de comunicação do MIMO, bem como a implementação da máquina de estados de controle do monitor.

6.5 PROGRAMA DE CONTROLE DO MONITOR

O objetivo do programa de controle do monitor é prover um meio para que o usuário tenha acesso, comande a operação e leia dados de rastreamento do monitor.

O programa de controle do monitor foi escrito em linguagem ANSI C para executar em um ambiente compatível com o padrão POSIX (*Portable Operating System Interface*). Ele necessita de que o computador hospedeiro disponibilize uma porta serial padrão RS-232, para a comunicação com o monitor. Sobre a linha física RS-232, o programa implementa o protocolo do MIMO, conforme descrito no Anexo II.

A interface entre o programa de controle do monitor e o usuário é feita em linha de comando. Ao invocar o programa, o usuário tem ao seu dispor nove comandos diferentes, conforme lista apresentada na Tabela 10.

Tabela 10: Comandos disponíveis no programa de controle do monitor.

Comando	Significado
`?` ou `help`	Apresenta informações sobre os comandos disponíveis.
`alive`	Envia ao monitor um comando de ecoar, informando ao usuário se o monitor está pronto para operar ou não.
`exit` ou `quit`	Encerra o programa.
`info`	Envia ao monitor um comando de ler configuração, apresentando ao usuário os dados lidos ou o motivo de o comando não haver sido completado.
`reset`	Envia ao monitor um comando de reiniciar.
`start`	Envia ao monitor o comando para iniciar amostragem, informando ao usuário se o comando foi confirmado pelo monitor ou não.
`stop`	Envia ao monitor o comando para finalizar amostragem, informando ao usuário se o comando foi bem sucedido ou não.
`upload`	Lê a configuração do monitor e, caso ele esteja parado e com eventos armazenados na memória, realiza a leitura dos eventos e monta um arquivo com as informações de rastreamento coletadas.
`version`	Informa a versão do programa de controle do monitor.

Utilizando os comandos apresentados na Tabela 10 o usuário é capaz de realizar todas as operações de monitoração.

6.6 FILTRAGEM E ANÁLISE

A monitoração de um sistema produz uma grande quantidade de dados. Para obter informações a partir destes dados é necessário realizar operações de filtragem e análise, conforme mostrado na Figura 14. O processo de filtragem e análise no SoftScope é realizado por programas de processamento, conforme mostrado na Figura 27.

As informações de entrada para o processo de filtragem e análise estão ao lado esquerdo da Figura 27, e são: o arquivo de rastreamento, a descrição das tarefas, a descrição da instrumentação e as restrições temporais. O arquivo de rastreamento é obtido pelo programa de controle do monitor. A descrição das tarefas é um arquivo fornecido opcionalmente pelo usuário para facilitar a compreensão e a visualização dos resultados. Ele contém o identificador, o nome e a prioridade de cada uma das tarefas do sistema. A descrição da instrumentação é um arquivo gerado pelo instrumentador, conforme descrito na Seção 6.2 e exemplificado na Figura 24. Finalmente, o arquivo das restrições temporais contém os atributos que as descrevem, conforme apresentado na Seção 5.4.3.

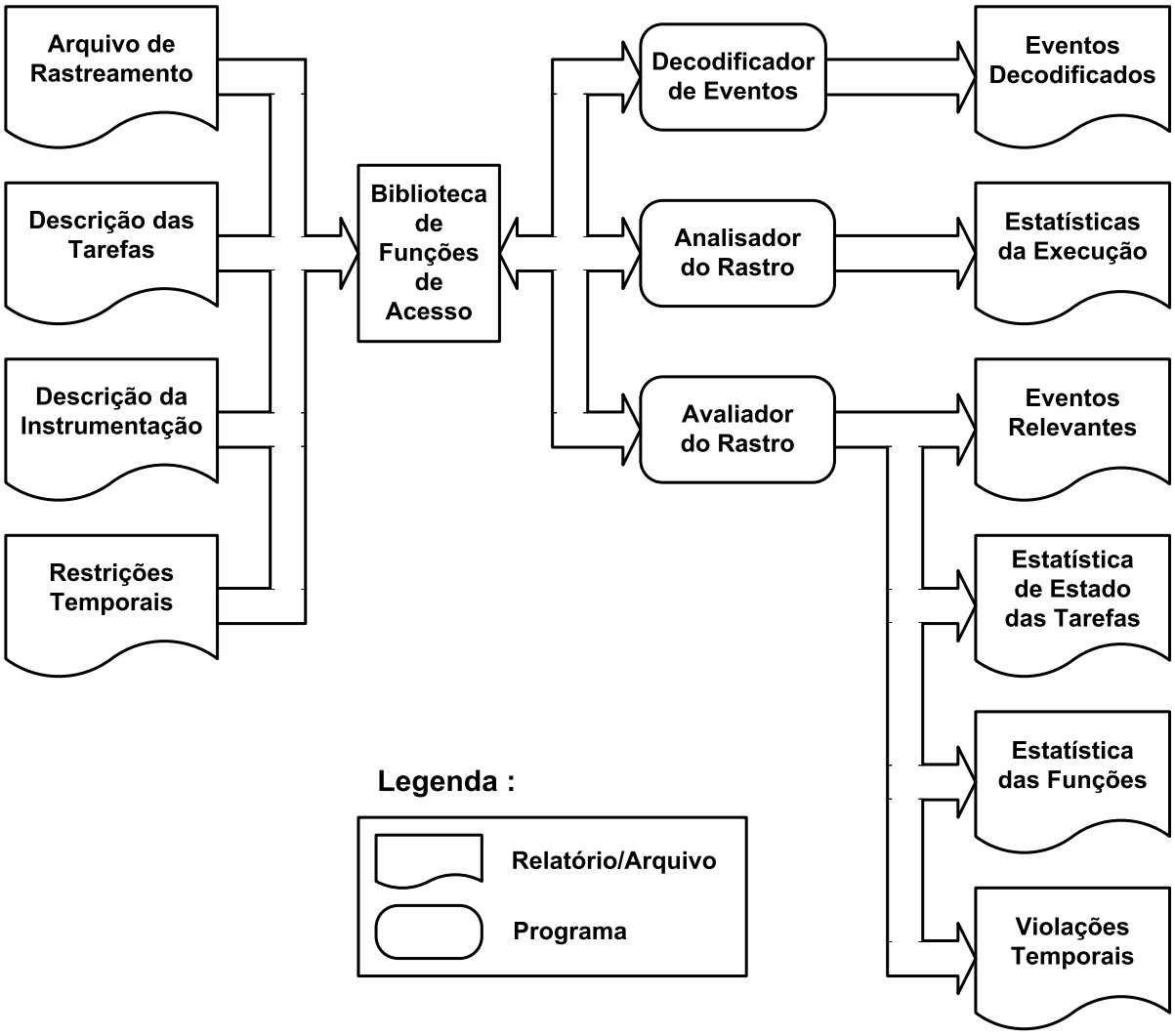


Figura 27: Processo de filtragem e análise do SoftScope.

Entre as informações de entrada e os programas de processamento, a Figura 27 mostra uma biblioteca de funções de acesso, cujo objetivo é fornecer aos programas uma visão uniforme das informações de entrada.

As operações de filtragem e análise são feitas por três programas: o decodificador de eventos, o analisador do rastro e o avaliador do rastro. O decodificador de eventos tem o objetivo de transformar o arquivo binário obtido pelo programa de controle do monitor em um arquivo texto contendo todos os eventos ocorridos durante a monitoração, em ordem seqüencial. Um exemplo de eventos decodificados por este programa é mostrado na Figura 28.

```
0001995200 [000000098:00000015]: Decrementer exception
0000005360 [000000106:00000016]: End of exception
0001995200 [000000112:00000017]: Decrementer exception
0000006340 [000000120:00000018]: Put MSG from ISR to "TimeManager"
0000010000 [000000128:00000019]: Schedule - running now is thread "TimeManager"
0000001280 [000000136:00000020]: End of exception
0000009340 [000000142:00000021]: System Call exception
0000006040 [000000150:00000022]: Receive Put MSG in thread "TimeManager"
0000007260 [000000158:00000023]: End of exception
0000006500 [000000164:00000024]: Timer expired for thread "ttyManager"
0000004840 [000000172:00000025]: System Call exception
0000007680 [000000180:00000026]: Put MSG from "TimeManager" to "ttyManager"
0000010280 [000000190:00000027]: End of exception
0000011040 [000000196:00000028]: System Call exception
0000005000 [000000204:00000029]: Receive MSG without any MSG - thread "TimeManager" blocked
0000005640 [000000212:00000030]: Schedule - running now is thread "ttyManager"
0000002020 [000000220:00000031]: End of exception
0000007800 [000000226:00000032]: System Call exception
0000006820 [000000234:00000033]: Receive Put MSG in thread "ttyManager"
0000006200 [000000242:00000034]: End of exception
0000004200 [000000248:00000035]: User event: (5)
0000032800 [000000254:00000036]: User event: (6)
```

Figura 28: Exemplo de saída do decodificador de eventos.

A primeira coluna de valores mostrada na Figura 28 refere-se ao tempo decorrido entre o evento corrente e o imediatamente anterior, em nano segundos. A seguir, os dois valores entre colchetes, separados por dois pontos, mostram o deslocamento do evento em número de bytes desde o início do arquivo de rastreamento e o número seqüencial do evento, respectivamente, seguidos de uma breve descrição.

O próximo programa de processamento mostrado na Figura 27 é analisador de rastro, cujo objetivo é obter do arquivo de rastreamento as estatísticas da execução, tomando como base os eventos do sistema operacional. Um exemplo de saída do analisador de rastro é mostrado na Figura 29.

```
Trace total time: 19.9s
CPU Idle time: 16.9s (84.66%)
CPU SYS time: 232.1ms (1.16%)
Threads time: 2.8s (14.17%)

Exceptions: "- Exception Name: [invocations], [OBCET], [OTCET], [OWCET];"
- Decrementer: <9965>, 5.3us, 7.3us, 23.2us;
- System Call: <10124>, 9.9us, 15.7us, 23.8us;

System services: "- System Service: [number_of_calls], [OBCET], [OTCET], [OWCET];"
- PutMessage   : <1116>, 14.5us, 17.8us, 19.4us;
- SendMessage  : <1156>, 17.4us, 22.5us, 23.8us;
- ReceiveMessage (empty)   : <2352>, 9.9us, 12.7us, 14.3us;
- ReceiveMessage (Put)     : <2192>, 11.9us, 13.1us, 14.9us;
- ReceiveMessage (no match): <996>, 16.7us, 17.0us, 17.6us;
- ReceiveMessage (Send)    : <1156>, 13.6us, 15.2us, 15.7us;
- ReplyMessage : <1156>, 14.2us, 17.0us, 19.1us;

Interrupt handling "- IRQ Name: [invocations], [OBCET], [OTCET], [OWCET];"

Threads: "- Thread Name: [number_of_schedules], [CPU_Time] (CPU_Time%);"
- TimeManager: <8728>, 70.5ms (0.35%);
- ttyManager: <4990>, 83.6ms (0.42%);
- Heartbeat(GREEN): <120>, 1.3ms (0.01%);
- Heartbeat(RED): <120>, 1.2ms (0.01%);
- CryptoManager: <202>, 118.1ms (0.59%);
- BlowfishEncrypter: <82680>, 1.3s (6.34%);
- BlowfishDecrypter: <82720>, 1.3s (6.37%);
```

Figura 29: Exemplo de saída do analisador de rastro.

Como se pode observar na Figura 29, a saída do analisador de rastro é composta por cinco seções, que são: sumário, estatística de exceções, estatística de chamadas de sistema, estatística de tratamento de interrupções e estatística de tarefas. O sumário é formado pela informação de quanto tempo de execução existe no arquivo de rastreamento, seguido de uma divisão deste tempo em três partes: o tempo em que o processador estava ocioso, o tempo utilizado na execução de chamadas de sistema e o tempo utilizado pelas tarefas. As outras quatro seções de saída possuem um cabeçalho e informações estatísticas. O cabeçalho reporta o tipo de informação que será encontrada naquela seção (exceções, chamadas de sistema, interrupções ou tarefas), seguido do formato de cada linha de saída. A estatística de exceções mostra quais exceções ocorreram durante a coleta dos dados, o número de ocorrências de cada exceção, o OBCET, o OTCET e o OWCET do tratamento. A estatística de chamadas de sistema reporta quais serviços do sistema operacional foram invocados, o número de chamadas de cada serviço, o OBCET, o OTCET e o OWCET. A estatística de tratamento de interrupções mostra que interrupções ocorreram, o número de ocorrências de cada interrupção, o OBCET, o OTCET e o OWCET do serviço de atendimento. Observe-se que no exemplo da Figura 29 nenhuma interrupção externa foi detectada durante a coleta de dados, fazendo com que a seção correspondente tenha apenas cabeçalho. Finalmente, a

estatística de tarefas mostra o nome de cada tarefa executada durante o rastreamento, o número de escalonamentos de cada tarefa pelo sistema operacional, o tempo de execução absoluto e o tempo de execução percentual. Com o analisador de rastro é possível identificar faltas de concorrência causadas por preempção e faltas de concorrência causada por interrupções (Seção 5.4.1), bem como faltas de processamento das tarefas (Seção 5.4.2).

O último programa de processamento é o avaliador do rastro, cujo objetivo é aprofundar a análise feita pelo analisador do rastro, obtendo informações que permitam fazer uma avaliação temporal detalhada. O avaliador do rastro gera quatro arquivos de saída: uma lista dos eventos relevantes para a reconstrução da linha de execução das tarefas, a estatística dos estados internos de cada uma das tarefas, a estatística das funções monitoradas e a lista das violações temporais. A lista dos eventos relevantes é um subconjunto de todos os eventos, só que tratados de modo a informar como reconstruir os caminhos de execução das tarefas. Esta lista é utilizada na construção de gráficos que permitem ao usuário visualizar as interações entre as tarefas e identificar os problemas de maneira mais simples. Um exemplo de um trecho de lista como esta é mostrado na Figura 30.

```
0000102319740, [000002426:00000326], SCHEDULE, 0, IdleTask, 0, 0, READY, RUNNING;
0000122042580, [000002566:00000346], PUT, -1, ISR, 1, 0, NONE, NONE;
0000122042580, [000002566:00000346], UNBLOCK, 1, TimeManager, 0, 0, RX BLOCKED, READY;
0000122053260, [000002574:00000347], PREEMPTION, 0, IdleTask, 0, 0, RUNNING, READY;
0000122054540, [000002582:00000348], SCHEDULE, 1, TimeManager, 0, 0, READY, RUNNING;
0000122069860, [000002596:00000350], RECEIVE, 1, TimeManager, 0, 0, RUNNING, RUNNING;
0000122097680, [000002626:00000354], PUT, 1, TimeManager, 3, 0, RUNNING, RUNNING;
0000122097680, [000002626:00000354], UNBLOCK, 3, ttyManager, 0, 0, RX BLOCKED, READY;
0000122124700, [000002650:00000357], RECEIVE, 1, TimeManager, 0, 0, RUNNING, RX BLOCKED;
0000122132020, [000002666:00000359], SCHEDULE, 3, ttyManager, 0, 0, READY, RUNNING;
0000122146620, [000002680:00000361], RECEIVE, 3, ttyManager, 0, 0, RUNNING, RUNNING;
```

Figura 30: Exemplo de um trecho de lista de eventos relevantes.

A Figura 30 mostra um trecho em que o sistema estava ocioso e, a partir de uma mensagem vinda de uma interrupção de *hardware*, o gerente de tempo (TimeManager) é invocado e, por sua vez, envia uma mensagem para outra tarefa (ttyManager), que havia pedido para ser acordada depois de um certo tempo. A primeira informação disponível em cada linha é o tempo decorrido desde o início do rastreamento até o instante atual. A seguir, entre colchetes e separados por dois pontos, estão o deslocamento do evento em número de bytes desde o início do arquivo de rastreamento e o número seqüencial do evento, respectivamente. Estas duas informações também existem no arquivo de eventos decodificados. O terceiro argumento é o nome do evento, seguido pelo identificador e pelo nome da tarefa em cujo contexto a execução se encontra. Note-se que na segunda linha do

exemplo mostrado na Figura 30 o identificador da tarefa é o número `-1` e o nome da tarefa é `ISR`, significando que este é um evento que foi gerado dentro de uma interrupção de *hardware* e, portanto, fora do contexto de qualquer uma das tarefas. Depois do nome da tarefa, o sexto e o sétimo argumentos são valores cujo significado depende do evento em questão. Por exemplo, no caso do serviço de PutMessage do PET#, indicado por `PUT` nas linhas dois e sete da Figura 30, o primeiro valor é o identificador da tarefa de destino, enquanto o segundo valor não tem significado algum. Vale ressaltar que, mesmo quando estes valores não tem significado, o campo correspondente é colocado no arquivo de saída para facilitar a leitura e a interpretação deste arquivo posteriormente por um analisador léxico. Os dois últimos argumentos são o estado atual e o próximo estado da tarefa.

Uma característica da lista de eventos relevantes é que um único evento no arquivo de rastreamento pode gerar mais de um evento nesta lista. Isto pode ser observado no exemplo da Figura 30 nas linhas dois e três, onde uma interrupção de *hardware* envia uma mensagem assíncrona para uma tarefa, o que também faz com que a tarefa destino mude seu estado de bloqueada em recepção para pronta. Algo similar também é observado nas linhas sete e oito do exemplo.

A segunda saída gerada pelo programa avaliador do rastro é a estatística dos estados internos de cada uma das tarefas, que é calculada e apresentada para cada tarefa que teve atividade durante a execução monitorada. Com estas informações o usuário é capaz de identificar faltas de concorrência causadas por sincronização (Seção 5.4.1). A forma de apresentação das saídas geradas neste arquivo é mostrada na Figura 31.

```
Thread name: CryptoManager (preempted 40 times)
--> Time spent in the UNKNOWN state: 308.2ms (1.53%)
--> Time spent in the READY state: 14.7ms (0.07%)
--> Time spent in the RUNNING state: 237.7ms (1.18%)
--> Time spent in the RX BLOCKED state: 14.4s (71.56%)
--> Time spent in the TX BLOCKED state: 5.2s (25.66%)
```

Figura 31: Apresentação da estatística dos estados de uma tarefa.

Na Figura 31 observa-se que um registro começa com o nome da tarefa e com o número de preempções que ela sofreu, ou seja, pelo número de vezes que a tarefa mudou do estado executando para o estado pronto. A seguir, cinco linhas mostram as estatísticas em cinco estados diferentes: desconhecido, pronto, executando, bloqueado em transmissão e bloqueado em recepção. Uma tarefa fica em estado desconhecido quando, no início do arquivo de rastreamento, ainda não é possível determinar seu estado. A partir do momento em que é possível determiná-lo, a tarefa fica em um dos quatro estados possíveis definidos pelo PET#, conforme diagrama de estados da Figura 52, Anexo I.

A estatística de estado das tarefas também pode ser mapeada em faltas de sincronização, computação e escalonamento do modelo de falta do VDS (Seção 4.4.2). Utilizando este outro modelo de falta, pode-se afirmar que as faltas de sincronização no PET# ocorrem quando uma tarefa passa muito tempo esperando por uma mensagem ou por uma resposta, ou seja, muito tempo no estado bloqueado em recepção ou no estado bloqueado em transmissão. Uma falta de computação ocorre quando uma tarefa passa muito tempo no estado executando, podendo consumir mais tempo de processamento do que o esperado e provocando a perda de prazo de outras tarefas menos prioritárias. Já uma falta de escalonamento ocorre quando uma tarefa passa muito tempo no estado pronto sem receber o processador. Assim como as faltas de concorrência e de processamento do modelo de falta do Dyretiva, a identificação de faltas no modelo de falta do VDS deve ser feita pelo usuário do SoftScope.

A terceira saída gerada pelo programa avaliador do rastro é a estatística das funções monitoradas, que é calculada e apresentada para cada função instrumentada que tenha sido referenciada durante a monitoração. Esta saída é capaz de auxiliar na identificação de faltas de processamento em funções (Seção 5.4.2). A forma de apresentação da estatística de funções monitoradas é mostrada na Figura 32.

```
Function [BlowfishEncrypt] in file [blowfish.c] has been executed [81920] times
--> Average execution time   : 29.5us
--> Best  case execution time: 26.7us ([BlowfishEncrypter], trace offset [1950284], event# [312824])
--> Worst case execution time: 46.5us ([BlowfishEncrypter], trace offset [2193614], event# [351880])
Function [BlowfishDecrypt] in file [blowfish.c] has been executed [81920] times
--> Average execution time   : 29.6us
--> Best  case execution time: 26.9us ([BlowfishDecrypter], trace offset [173538], event# [27856])
--> Worst case execution time: 45.8us ([BlowfishDecrypter], trace offset [589124], event# [94522])
```

Figura 32: Apresentação da estatística de execução das funções monitoradas.

Na Figura 32 são mostradas as estatísticas de duas funções monitoradas: BlowfishEncrypt e BlowfishDecrypt. Cada estatística de função monitorada possui um cabeçalho com o nome da função, o nome do arquivo fonte onde o código da função se encontra e o número execuções observadas durante a monitoração. Na linha seguinte é mostrado o tempo médio de execução da função durante o rastreamento, obtido pela média aritmética de todas as instâncias encontradas durante a monitoração. Na próxima linha é apresentado o tempo de execução da função observado no melhor caso, que é acompanhado da informação da tarefa na qual ocorreu este melhor caso e da identificação da instância que o gerou dentro do arquivo de rastreamento. A última linha de cada entrada informa o tempo de execução na função observado no pior caso, que também é acompanhado da informação da tarefa na qual ocorreu este pior caso e da identificação da instância do evento dentro do arquivo de rastreamento.

A análise das funções monitoradas no SoftScope leva em consideração o contexto onde a função está executando, e também o fato dela poder ser interrompida por preempção. Aliás, os piores casos de tempo de execução das funções monitoradas mostradas na Figura 32 aconteceram justamente quando a função foi interrompida por preempção, o que fez com que a parte do tempo de comutação de contexto executada ainda no espaço do usuário fosse contada como tempo de execução da função.

A quarta saída gerada pelo programa avaliador do rastro é o relatório das violações temporais, que é gerado a partir das restrições temporais fornecidas pelo usuário. Este relatório identifica as faltas de prazo (Seção 5.4.3). A Figura 33 mostra um exemplo de arquivo contendo restrições temporais que o usuário deve fornecer para o avaliador de rastro, enquanto a Figura 34 mostra um exemplo de relatório de violações temporais geradas pelo avaliador de rastro.

A sintaxe do arquivo de restrições temporais requer que cada declaração seja feita em uma linha e que esteja entre colchetes, conforme mostrado no exemplo da Figura 33. A primeira linha deve informar quantas restrições temporais (Seção 5.4.3) serão descritas no arquivo. Em seguida, cada restrição temporal é descrita individualmente, iniciando com a atribuição de um identificador, que será utilizado em futuras referências àquela restrição. Em seguida, são informados o evento inicial e o evento final da restrição e que são descritos por cinco argumentos cada, em formato compatível com o modelo do Dyretiva (Seção 5.3.4). O primeiro argumento é o tipo do evento, que pode ser evento do usuário (`USER`) ou do sistema operacional (`OS`). O próximo argumento indica quantas escritas adicionais na porta de monitoração serão utilizadas para descrever o evento em questão. Este valor será igual a zero

no caso de evento do usuário, ou de até dois no caso de evento do sistema operacional. O terceiro argumento é o identificador do evento, que combinado com o tipo dará o padrão de busca no arquivo de rastreamento. Os dois argumentos finais são os valores das escritas adicionais na porta de monitoração, quando aplicável. A próxima parte da descrição da restrição temporal informa as tarefas envolvidas e começa com a indicação do número de tarefas envolvidas, seguida do nome de cada tarefa. Finalmente, a última parte da descrição de uma restrição temporal informa o prazo e o período da restrição em quantidades de milésimos de segundos. Deve existir uma descrição de restrição temporal no arquivo para cada restrição temporal identificada pela Engenharia de Requisitos.

```
[NUMBER_OF_TIME_REQUIREMENTS = 2]
[TIME_REQUIREMENT_NAME = UmaRestricaoTemporal]
[START_EVENT = OS, 2, 82, 9, 10]
[END_EVENT = OS, 2, 87, 11, 9]
[RELATED_THREADS = 2]
[THREAD1 = NomeDaPrimeiraTarefaEnvolvida]
[THREAD2 = NomeDaSegundaTarefaEnvolvida]
[DEADLINE = 300]
[INTERVAL = 500]
[TIME_REQUIREMENT_NAME = OutraRestricaoTemporal]
[START_EVENT = OS, 2, 82, 9, 12]
[END_EVENT = OS, 2, 87, 13, 9]
[RELATED_THREADS = 1]
[THREAD1 = NomeDaTarefaEnvolvida]
[DEADLINE = 200]
[INTERVAL = 1000]
```

Figura 33: Exemplo de arquivo de restrições temporais.

```
Time requirement [UmaRestricaoTemporal]
Deadline  [300ms]
Period    [500ms]
Instances [40]
Average jitter    [150.2us]
Worst case jitter [210.1us]
Missed deadlines [2]
--> 1st missed deadline offset         [1872890]
--> 1st missed deadline event number   [305587]
--> 1st missed deadline total execution time [301.2ms]
--> 1st missed deadline feature execution time [296.1ms]
--> 1st missed deadline feature blocked by the system [3.2ms]
--> 1st missed deadline feature blocked by other threads [1.9ms]
Met deadline in the best case [275.8ms]
--> Met deadline best case offset [336138]
--> Met deadline best case event number [54707]
Met deadline in the worst case [298.9ms]
--> Met deadline worst case offset [1982658]
--> Met deadline worst case event number [323507]
```

Figura 34: Exemplo de relatório de violações temporais.

A Figura 34 mostra um exemplo de seção de saída no relatório de violações temporais. O relatório inicia informando o nome da restrição temporal, o prazo e o período. Logo após é informado o número de instâncias desta restrição temporal encontrados no arquivo de rastreamento, seguido do desvio de período médio e do desvio de período no pior caso (*jitter*). A parte seguinte mostra se a restrição temporal foi obedecida e, em caso negativo, são dados detalhes da primeira violação encontrada. Como parte das informações referentes à violação temporal, o avaliador do rastro diz o tempo entre o evento inicial e o evento final, e também quanto deste tempo foi utilizado pelo sistema operacional e por outras tarefas. Com estas três informações o usuário pode inferir se a falta correspondente foi de processamento (Seção 5.4.2) ou de concorrência (Seção 5.4.1). A seguir o avaliador do rastro ainda fornece duas informações relevantes para a análise temporais: o melhor caso e o pior caso no tempo de execução que atendem a restrição temporal.

Em resumo, com as operações de filtragem e de análise dos dados de rastreamento o SoftScope é capaz de obter informações sobre o comportamento dinâmico do SUT, conforme o modelo de utilização, a abordagem de monitoração, e o modelo de falta estabelecidos pelo método Dyretiva.

6.7 APRESENTAÇÃO DOS RESULTADOS

As camadas de filtragem e de análise geram como resultado diferentes listagens contendo as informações necessárias para avaliar o SUT. Para centralizar e tornar mais fácil a interpretação destas listagens o SoftScope utilizou uma ferramenta gráfica, adaptada do trabalho de Navarro [Navarro 06], cujo objetivo é proporcionar uma maneira de melhorar a compreensão do comportamento dinâmico de sistemas concorrentes. Esta ferramenta foi escrita em linguagem de programação C#, e seu ambiente de execução é o *.NET framework* da Microsoft. Na adaptação feita para o SoftScope, quatro formas de apresentação de resultados foram previstas: janelas informativas, gráficos de setores, tabelas e gráficos e Gannt. A janela principal do programa de apresentação de resultados é mostrada na Figura 35.

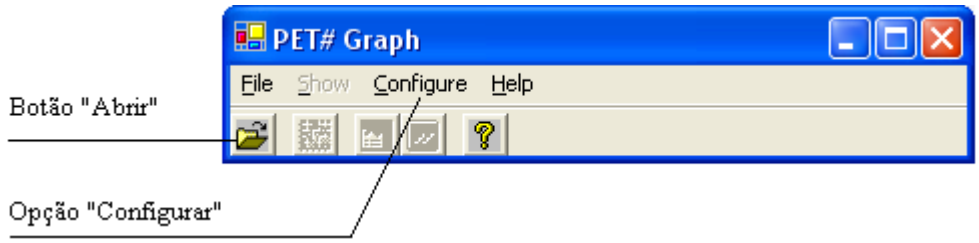


Figura 35: Janela principal do programa de apresentação dos resultados.

O programa de apresentação recebe as mesmas quatro entradas fornecidas para as camadas de filtragem e de análise, conforme mostrado no processo da Figura 27. Isto pode ser feito, de acordo com a interface gráfica mostrada na Figura 35, utilizando o botão "Abrir" para entrar com o arquivo de rastreamento a ser analisado. Já pela opção "Configurar" é possível informar os arquivos que serão utilizados para descrever as tarefas, a instrumentação e as restrições temporais. A partir daí, conforme a solicitação feita pelo usuário, o programa de apresentação invoca os programas de processamento, lê e interpreta as listagens de saída geradas por eles, e providencia uma forma adequada mostrar os resultados para o usuário.

Após as entradas serem fornecidas ao programa de apresentação, os três botões de solicitação de informações são habilitados, conforme mostrado na Figura 36.

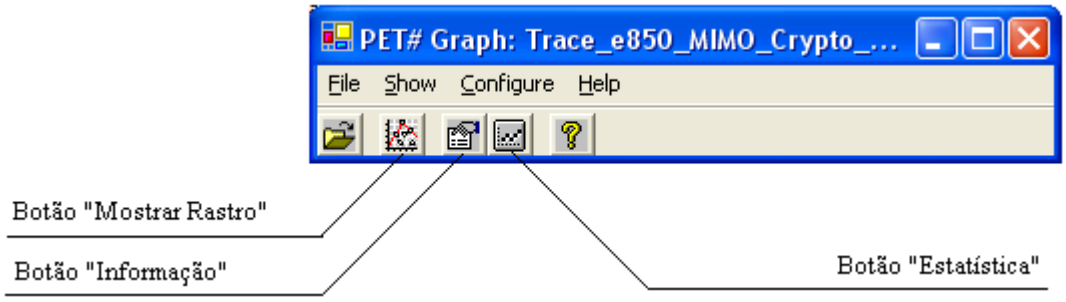


Figura 36: Programa de apresentação com botões de apresentação habilitados.

O botão “Informação”, quando pressionado, apresenta ao usuário uma janela informativa com informações tiradas do cabeçalho do arquivo de rastreamento. Uma janela como esta pode ser vista na Figura 37, cuja primeira linha de informação contém a localização e o nome do arquivo de rastreamento. A segunda linha mostra a ordenação de bytes do conteúdo do arquivo de rastreamento, que pode ser *big endian* ou *little endian*. A terceira linha informa o número de bytes de dados contidos no arquivo de rastreamento, excluindo o tamanho do cabeçalho, enquanto a quarta linha informa quantos eventos existem no número de bytes informados na linha anterior. Esta duas informações são complementares, uma vez que os eventos possuem tamanho variável. A quinta linha, que trata da posição de *trigger* dentro do arquivo de rastreamento, está reservada para uso futuro. Finalmente, a última linha apresenta um texto descritivo contido no cabeçalho do arquivo.

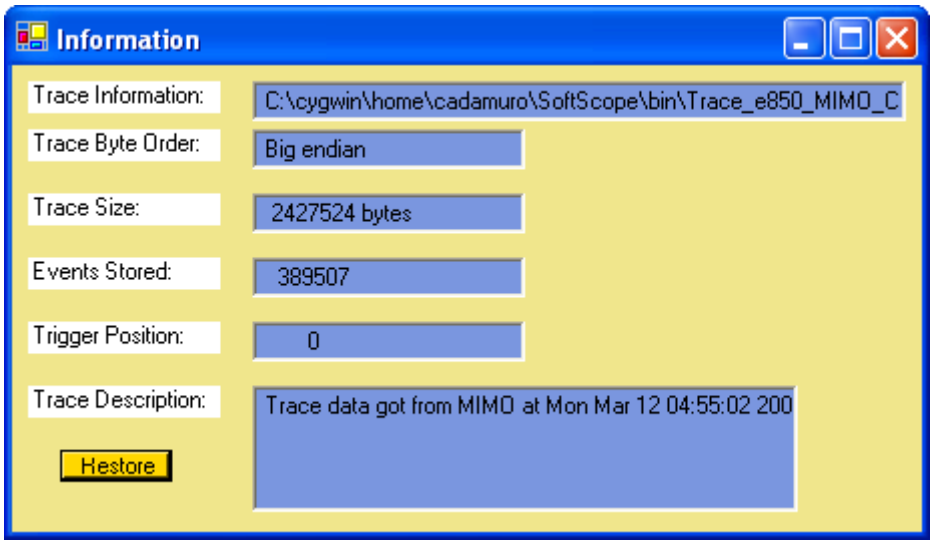


Figura 37: Janela de informação do programa de apresentação.

Ao pressionar o botão “Estatística” na interface gráfica da Figura 36 o usuário recebe uma janela informativa contendo o sumário da execução monitorada, conforme pode ser visto na Figura 38. Os dados do sumário são gerados pelo programa analisador do rastro, apresentado na seção anterior. Se o usuário pressionar o botão “Grafo” desta janela, receberá dois gráficos de setores com informações sobre o processador, conforme pode ser visto na Figura 39.

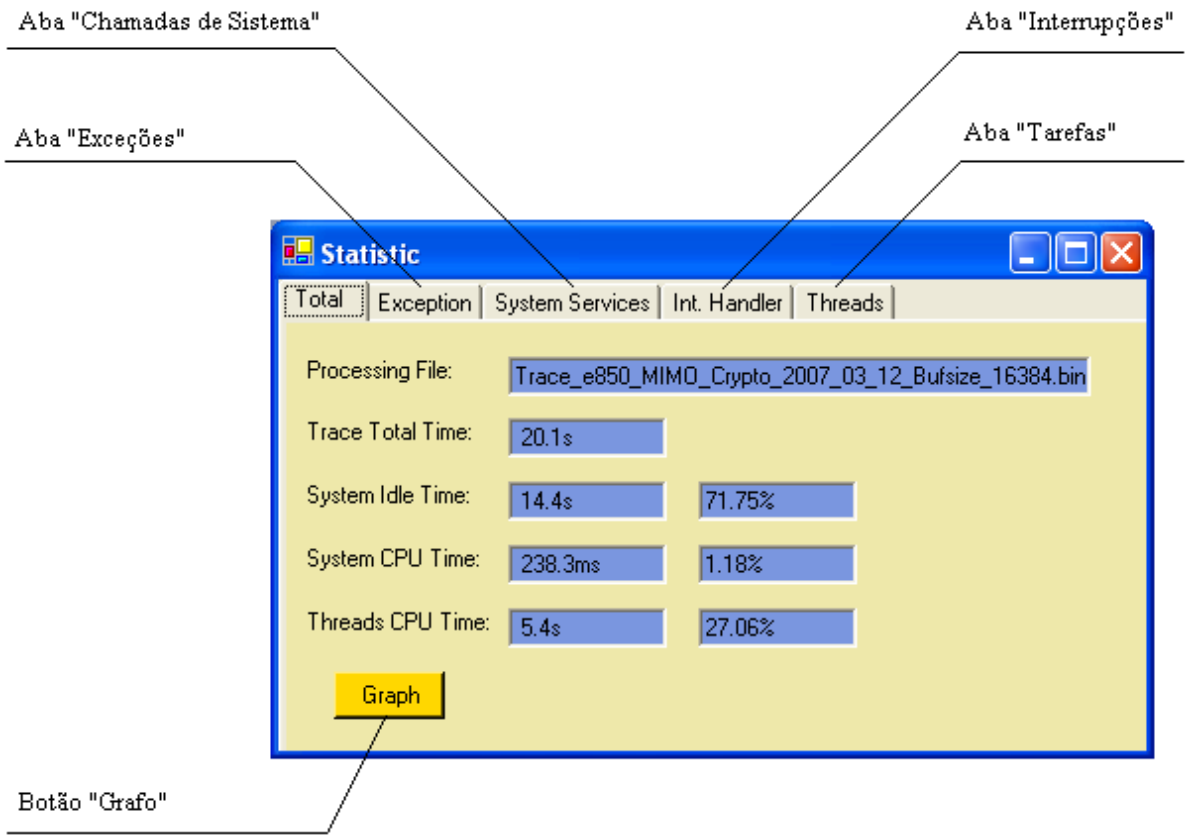


Figura 38: Apresentação dos dados estatísticos da execução monitorada.

O gráfico de setores na parte superior da Figura 39 mostra a divisão do tempo do processador entre ocioso, sistema e tarefas. Já o gráfico de setores na parte inferior da Figura 39 mostra como foi o tempo total destinado às tarefas foi subdividido entre as tarefas ativas durante a monitoração.

Além do sumário da execução e dos gráficos de setores referentes ao uso do processador, a apresentação dos dados estatísticos também é capaz de mostrar, na forma de tabelas, as outras informações geradas pelo programa analisador do rastro. Estas informações ficam disponíveis nas abas “Exceções”, “Chamadas de Sistema”, “Interrupções” e “Tarefas”, indicadas na Figura 38. Ao pressionar a aba “Tarefas”, por exemplo, o usuário recebe a tabela contendo as informações referentes ao tempo de execução das tarefas, como mostrado na Figura 40. O uso de tabelas, ao invés dos relatórios de texto gerados pelo analisador de rastro, permite que o usuário organize os dados, podendo ordená-los de forma crescente ou decrescente pelo conteúdo de qualquer uma das colunas.

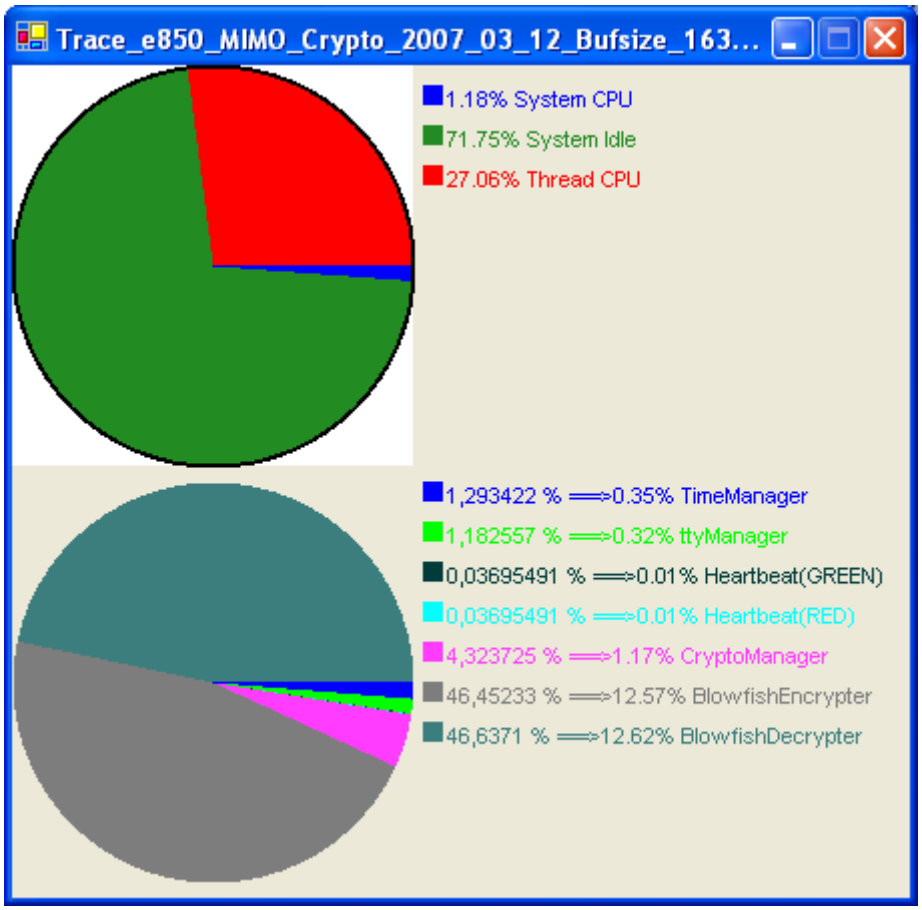
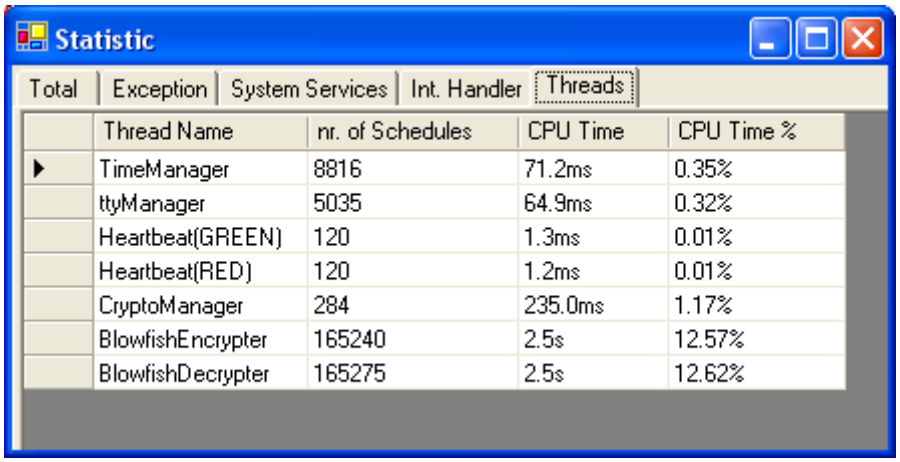


Figura 39: Gráfico de setores da utilização do processador.



	Thread Name	nr. of Schedules	CPU Time	CPU Time %
▶	TimeManager	8816	71.2ms	0.35%
	ttyManager	5035	64.9ms	0.32%
	Heartbeat(GREEN)	120	1.3ms	0.01%
	Heartbeat(RED)	120	1.2ms	0.01%
	CryptoManager	284	235.0ms	1.17%
	BlowfishEncrypter	165240	2.5s	12.57%
	BlowfishDecrypter	165275	2.5s	12.62%

Figura 40: Tabela do tempo de execução das tarefas.

Finalmente, ao pressionar o botão “Mostrar Rastro” na interface gráfica da Figura 36 o usuário pode acessar as informações geradas pelo programa avaliador do rastro e visualizar o comportamento dinâmico do sistema. A janela principal de apresentação do rastro é mostrada na Figura 41 e contém a tabela dos eventos relevantes para a reconstrução da linha de execução das tarefas. A aba “Todos os Eventos” apresenta uma tabela com a saída gerada pelo programa decodificador de eventos. As abas “Estatística de Estado das Tarefas” e “Estatística de Funções” contém tabelas com informações geradas pelo programa avaliador do rastro. A aba “Prazos Perdidos” traz informações sobre a avaliação temporal das tarefas gerada pelo programa avaliados do rastro. Finalmente na aba “Grafo” o programa de apresentação possibilita a visualização do comportamento dinâmico do sistema utilizando um gráfico de Gannt. O conteúdo desta aba pode ser visto na Figura 42.

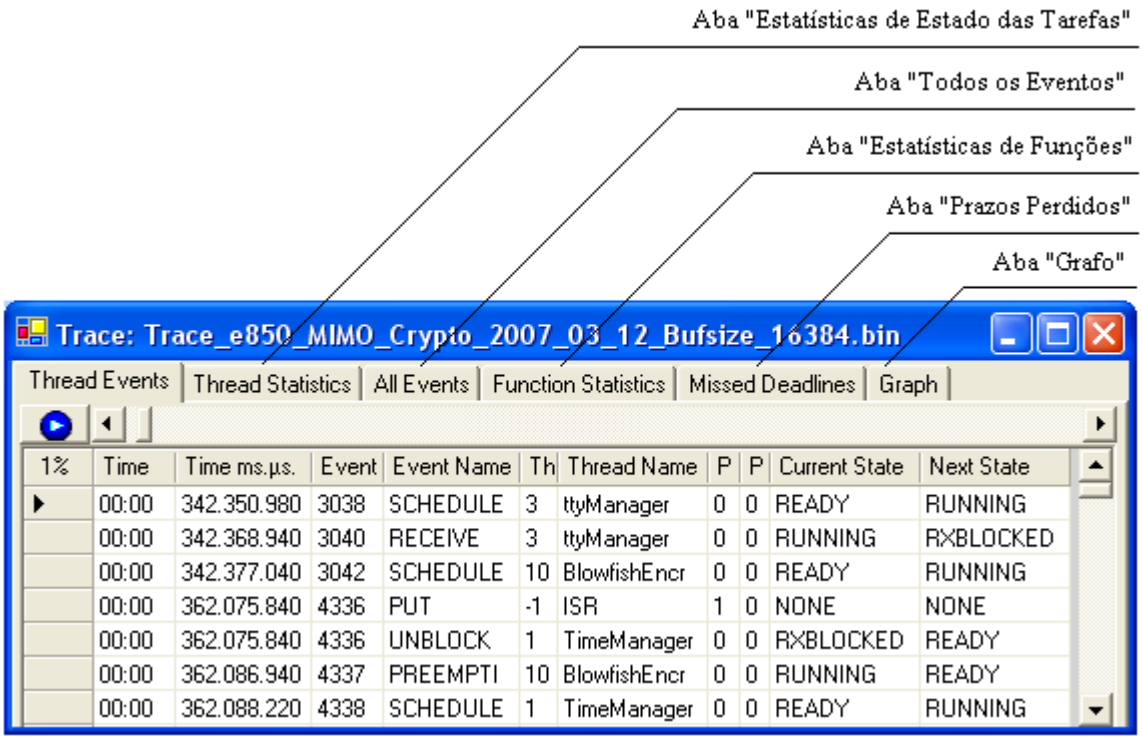


Figura 41: Janela de apresentação do rastro.

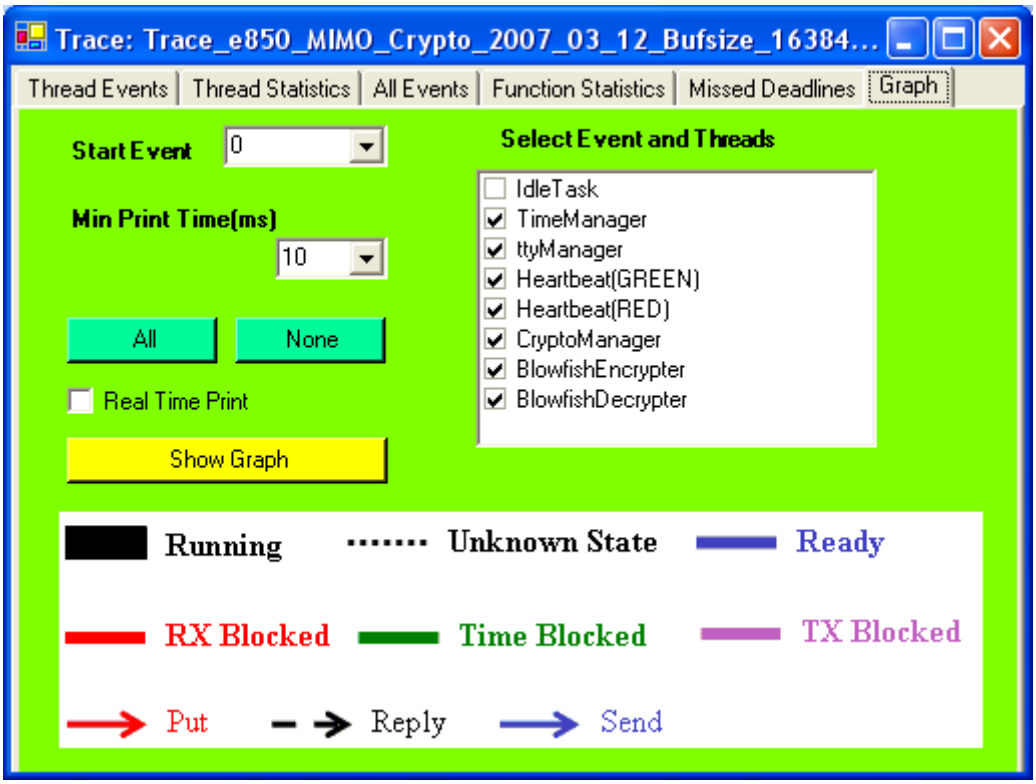


Figura 42: Conteúdo da aba "Grafo" na apresentação do rastro.

Na Figura 42 podem ser destacados três pontos principais. Primeiro, utilizando a lista das tarefas do lado direito da janela é possível escolher os eventos de quais tarefas serão apresentados no gráfico de Gannt. Segundo, a opção de imprimir em proporção temporal, encontrada logo acima do botão amarelo do lado esquedo da janela, exige que o gráfico de Gannt seja desenhado guardando uma proporção para os intervalos do rastro. Se esta opção não for selecionada, então o gráfico será desenhado com eventos visualmente eqüidistantes. Terceiro, na parte inferior da janela existe uma legenda com o código de cores utilizado para representar os estados das tarefas e as diferentes formas de comunicação e sincronização por troca de mensagens. Uma vez que as opções de visualização desejadas tenham sido escolhidas, ao pressionar-se o botão amarelo "Mostrar Grafo" o gráfico de Gannt é desenhado, conforme exemplo mostrado na Figura 43.

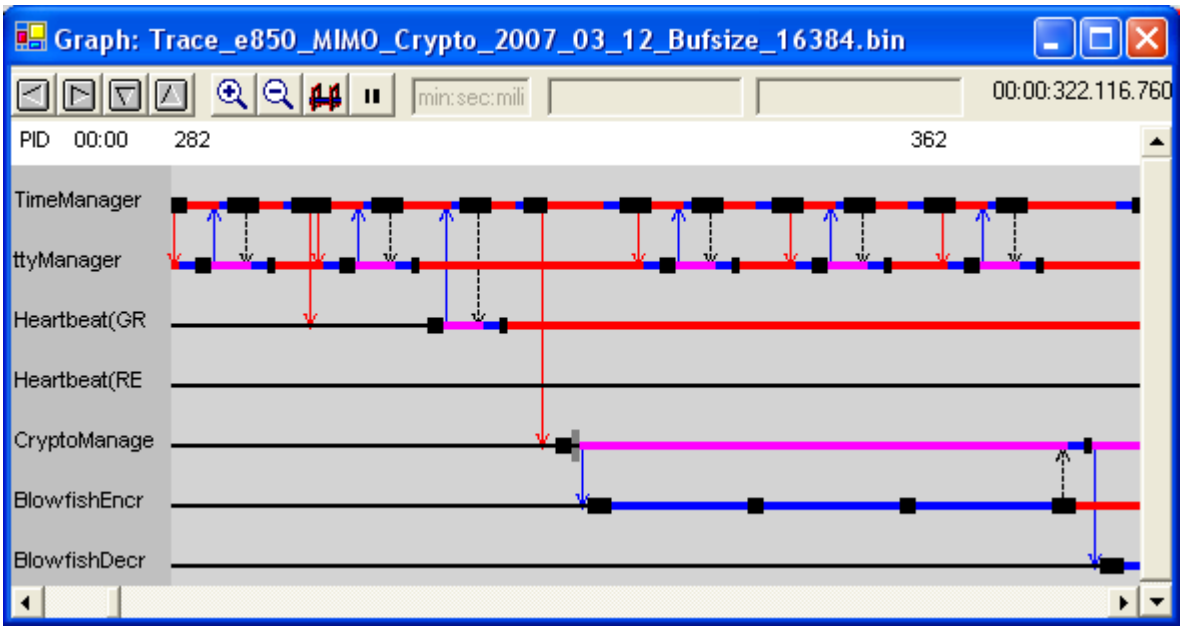


Figura 43: Exemplo de gráfico de Gannt.

Em resumo, com o programa de apresentação dos resultados o SoftScope é capaz de mostrar as informações obtidas pelos programas de processamento de forma mais amigável, e também provê uma forma de visualizar o comportamento dinâmico do sistema através de gráficos de Gannt.

6.8 RESUMO

Neste capítulo foram apresentadas as ferramentas que compõe o SoftScope, que são: o pré-instrumentador de código fonte C, o instrumentador de código fonte C, o monitor híbrido externo (MIMO), o programa de acesso ao monitor, os programas de processamento dos dados coletados, e o programa de apresentação dos resultados.

O pré-instrumentador e o instrumentador são compiladores simplificados que auxiliam o usuário a colocar no seu código fonte as instruções de instrumentação necessárias para efetuar a monitoração.

O MIMO é a implementação de um monitor híbrido de acordo com os requisitos do método Dyretiva. Para os sistemas onde a monitoração híbrida não é possível, o SoftScope também permite o uso de um monitor baseado em *software*, apesar deste não possuir as características de precisão e pouca intrusão necessárias na monitoração de sistemas em tempo real. Entretanto, é possível utilizá-lo como auxílio na identificação de problemas que ocorram nas fases iniciais de teste, compreendendo a monitoração com o objetivo de teste e depuração.

O programa de controle do monitor é uma ferramenta que permite que o usuário interaja com o MIMO, enviando comandos e solicitando as informações referentes a rastreamentos efetuados.

Os programas de processamento dos dados coletados, ou programas de filtragem e análise dos dados, realizam o trabalho de retirar dos dados as informações necessárias sobre o comportamento dinâmico do sistema. Após o seu trabalho, as informações de conformidade ou não do sistema já estão disponíveis, só que na forma de relatórios.

O programa de apresentação dos resultados utiliza uma interface gráfica para mostrar ao usuário as informações obtidas pelos programas de processamento dos dados de forma mais amigável. A apresentação pode ser feita na forma de janelas informativas, gráficos de setores, tabelas ou gráficos de Gannt, dependendo do tipo da informação.

Para verificar o funcionamento do Dyretiva e do SoftScope na obtenção das informações desejadas em termos de comportamento dinâmico e correção temporal de um sistema embarcado operando em tempo real, o próximo capítulo apresenta situações reais de uso, onde a eficácia do método e das ferramentas pode ser comprovada.

7 VALIDAÇÃO DO MÉTODO

Neste capítulo o Dyretiva e o SoftScope são utilizados em situações reais para descrever o comportamento dinâmico de um sistema embarcado e verificar o cumprimento de restrições temporais.

Nas seções a seguir é apresentado o ambiente de teste, a implementação sob teste e os casos de teste utilizados. Durante a execução dos casos de teste o sistema é monitorado e os dados de rastreamento são coletados. Utilizando as ferramentas do SoftScope estes dados são processados e analisados, fornecendo informações de desempenho do sistema, seu comportamento dinâmico e temporal. Ao final dos experimentos também é feita uma avaliação da intrusão causada pela instrumentação do Dyretiva que permitiu a coleta dos dados, verificando sua viabilidade para aplicações de tempo real.

7.1 AMBIENTE DE TESTE

O SUT utilizado para validar o método Dyretiva e fornecer os dados necessários para exercitar as ferramentas do SoftScope é composto do módulo e850Lite (Seção 4.1) executando o sistema operacional PET# (Seção 4.3 e Anexo I) e uma aplicação de teste. O SUT pode fornecer dados de monitoração de duas formas diferentes: através do MIMO (Seção 6.3) ou através de monitoração baseada em *software* (Seção 6.4).

No SUT a monitoração baseada em *software* é possível porque o módulo e850Lite reúne as condições de *hardware* necessárias, ou seja, uma porta serial RS-232, espaço em memória e capacidade de processamento. Um diagrama em blocos do ambiente de teste, quando utilizando monitoração baseada em *software*, é mostrado na Figura 44. Neste diagrama em blocos o módulo e850Lite é apresentado ressaltando seus principais componentes e interfaces externas. O quadrado pontilhado dentro do bloco e850Lite com a inscrição “Monitor” ressalta o fato de que, neste caso, a monitoração está sendo feita a partir de blocos de *hardware* e de *software* do próprio SUT. Pode-se observar também na figura que as duas portas seriais RS-232 do módulo, indicadas por COM1 e COM2, estão em uso. A primeira está conectada a um emulador de terminal, tipo VT-100, através do qual o PET# apresenta um interpretador de comandos ao usuário para que este interaja com o sistema operacional e com a aplicação. A segunda porta serial está ligada a um computador que possui o SoftScope instalado, permitindo a comunicação com o programa de controle do monitor.

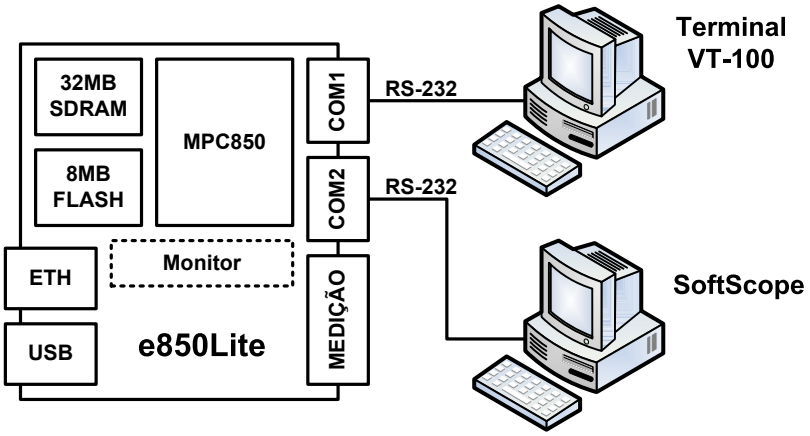


Figura 44: Ambiente de teste com monitoração baseada em *software*.

A outra forma do SUT fornecer dados para rastreamento é através da porta de monitoração, ligada ao monitor externo (MIMO) por um cabo plano. Este é o caso de operação preferido do Dyretiva, pois a intrusão causada é muito menor do que no caso anterior de monitoração baseada em *software*. Um diagrama em blocos do ambiente de teste, quando utilizando monitoração híbrida, é mostrado na Figura 45.

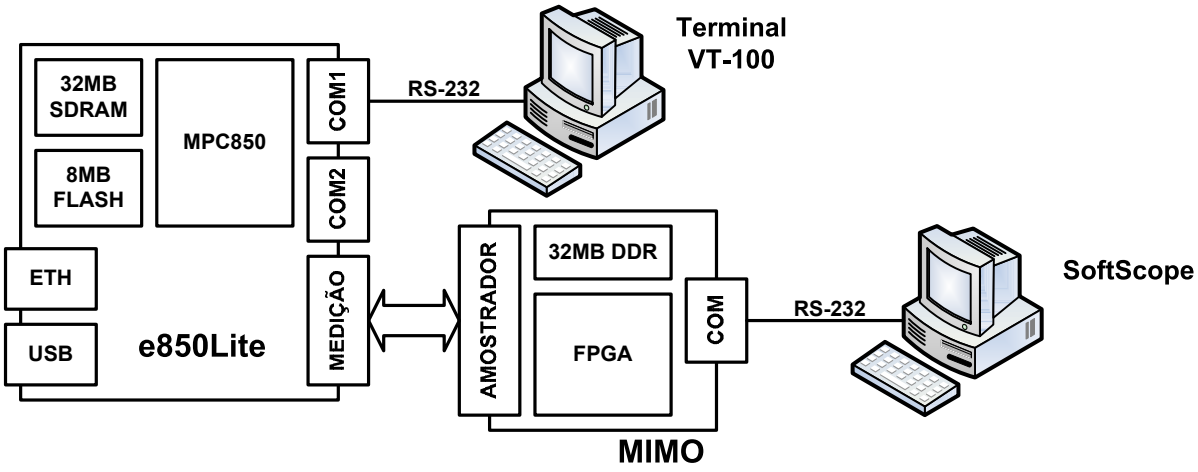


Figura 45: Ambiente de teste com monitoração híbrida.

O cenário mostrado na Figura 45 é diferente do cenário mostrado na Figura 44 por dois aspectos principais. Primeiro, o bloco em linhas pontilhadas com a inscrição “Monitor” dentro do diagrama da e850Lite foi retirado, sugerindo que a monitoração híbrida consome menos recursos do SUT do que a monitoração baseada em *software*. Isto não significa que a intrusão no sistema é nula, pois instruções de instrumentação precisam continuar a ser escritas pelo SUT na porta de medição, só que, neste caso, não há nenhum tipo de processamento ou armazenamento pelo SUT. Segundo, o SoftScope não se comunica

diretamente com o SUT, mas, ao invés disso, com o MIMO. Deste modo, toda a parte de comunicação ligada à monitoração também é transferida do SUT para o monitor externo. Em todos os experimentos deste capítulo foi utilizada a monitoração híbrida como forma de coleta dos dados de rastreamento.

7.2 APLICAÇÃO DE TESTE

A aplicação de teste é uma das partes do ambiente de teste, também chamada de IUT. O objetivo da IUT neste trabalho é impor ao SUT diferentes cargas de processamento para que as características funcionais e temporais possam ser avaliadas pelo Dyretiva. A IUT escolhida foi uma simulação de canal de comunicação serial cujos dados estão criptografados pelo algoritmo Blowfish [Schneier 94]. Neste caso, variando a taxa do canal é possível gerar diferentes cargas de utilização da CPU. Além disso, duas características da criptografia mantêm semelhanças com os STR: o tempo de CPU é proporcional ao tamanho dos dados fornecidos e a quantidade de memória necessária para realizar as operações é conhecida e limitada. O conhecimento a priori da quantidade de memória necessária torna possível fazer alocação estática de memória e remove a possibilidade de distorção dos resultados temporais da análise em conseqüência das anomalias que podem resultar da alocação dinâmica.

A aplicação de teste é composta por doze tarefas, que são apresentadas na Figura 46 por uma tela do PET#. Esta ilustração mostra a mensagem que o interpretador de comandos envia para o usuário pela porta serial do módulo e850Lite. Na primeira linha da mensagem está a versão da biblioteca do núcleo e a data de sua compilação, seguida de algumas informações sobre o interpretador de comandos. Após esta mensagem inicial, uma linha de comando é disponibilizada ao usuário. No exemplo mostrado foi solicitado ao interpretador o comando `thread`, que tem a finalidade de mostrar no terminal todas as tarefas em execução pelo núcleo. O resultado do comando é apresentado em sete colunas. A primeira e a segunda colunas são o identificador da tarefa (`TID`) e o seu nome, respectivamente. A terceira coluna mostra o estado atual da tarefa, conforme diagrama de estados apresentado na Figura 52 do Anexo I. A quarta coluna mostra a prioridade da tarefa, que é um número de zero até trinta e um, sendo que quanto menor o número maior a prioridade da tarefa. Para o PET# duas tarefas prontas no mesmo nível de prioridade serão escalonadas pelo algoritmo FIFO. A quinta coluna mostra o número de mensagens assíncronas aguardando tratamento pela respectiva tarefa. A sexta coluna mostra o número de bytes que a tarefa possui em sua pilha privada e, na sétima coluna, quanto desta pilha está livre em termos percentuais.

```
PET# for MPC8xx, version 1.0.0 (Jul  8 2007 - 20:19:12)


*****************************************************************************
*                   PET# - A Preemptive Real-Time Microkernel                *
*                    By Joao Cadamuro Junior (CPGEI / UTFPR)                 *
*****************************************************************************
*                          Welcome to PET# Embedded Shell                     *
*                Type "help" to view the list of available commands           *
*****************************************************************************
[PET#]>
[PET#]>
[PET#]> thread

-----------------------------------------------------------------------
| TID | NAME                |   STATE    | PRIO | MSGs |  STACK | %FREE |
-----------------------------------------------------------------------
|   0 | IdleTask            | READY      |  31  |   0  |    512 |   64% |
|   1 | TimeManager         | RX_BLOCKED |   0  |   0  |   2048 |   86% |
|   2 | DeviceManager       | RX_BLOCKED |   1  |   0  |   2048 |   89% |
|   3 | ttyManager          | RX_BLOCKED |   7  |   0  |   2048 |   82% |
|   4 | EmbeddedShell#0     | READY      |  20  |   0  |   4096 |   73% |
|   5 | EmbeddedShell#1     | TX_BLOCKED |  20  |   0  |   4096 |   80% |
|   6 | Heartbeat(GREEN)    | RX_BLOCKED |  16  |   0  |   2048 |   86% |
|   7 | Heartbeat(RED)      | RX_BLOCKED |  25  |   0  |   2048 |   86% |
|   8 | FlashDriver#0       | RX_BLOCKED |   5  |   0  |   2048 |   88% |
|   9 | CryptoManager       | RX_BLOCKED |  22  |   0  | 133120 |    1% |
|  10 | BlowfishEncrypter   | RX_BLOCKED |  21  |   0  |   2048 |   88% |
|  11 | BlowfishDecrypter   | RX_BLOCKED |  21  |   0  |   2048 |   88% |
-----------------------------------------------------------------------

[PET#]>
```

Figura 46: Tarefas que compõe a aplicação de teste.

Das doze tarefas da IUT que estão executando no módulo e850Lite, seis pertencem ao sistema operacional, três são tarefas de suporte ao *hardware*, e três são tarefa da aplicação de criptografia. As seis tarefas do sistema são: IdleTask, TimeManager, DeviceManager, ttyManager, EmbeddedShell#0 e EmbeddedShell#1. A tarefa IdleTask é a menos prioritária do sistema e será selecionada para executar sempre que não houver uma outra tarefa pronta. O tempo de execução desta tarefa é o tempo ocioso do sistema. A tarefa TimeManager controla as temporizações das outras tarefas do sistema, acordando-as quando necessário. A tarefa DeviceManager implementa um gerente de dispositivos que permite o acoplamento ao núcleo de elementos que possam ser controlados utilizando cinco funções: `open`, `close`, `read`, `write` e `ioctl`. A tarefa ttyManager é um gerenciador de comunicações que implementa a abstração de um terminal TTY (Teletypewriter). Os programas que precisam se comunicar com o mundo exterior pelas portas de comunicação serial padrão RS-232 utilizam esta abstração para manipular a porta. Finalmente, as tarefas EmbeddedShell#0 e EmbeddedShell#1 implementam interpretadores de comandos do PET# e utilizam as portas seriais COM1 e COM2, via TTY, para se comunicar com o usuário. Como as duas portas

seriais estão em uso pelos interpretadores de comandos, a aplicação de teste está utilizando monitoração híbrida e enviando suas informações de monitoração pela porta de medição. Quando a monitoração baseada em *software* é utilizada, a tarefa EmbeddedShell#1 é removida e uma outra tarefa de emulação do MIMO é colocada em seu lugar, utilizando a porta COM2.

No próximo grupo, as três tarefas de suporte ao *hardware* são: Heartbeat(GREEN), Heartbeat(RED) e FlashDriver#0. Estas tarefas manipulam dispositivos físicos do módulo e850Lite. As duas primeiras tarefas manipulam dois LED (Light-Emitting Diode), um verde e outro vermelho, respectivamente, e a última tarefa implementa algoritmos e temporizações que permitem escrever e apagar dados de uma memória flash.

Finalmente, o último grupo é composto das três tarefas da aplicação, que são: CryptoManager, BlowfishEncrypter e BlowfishDecrypter. A tarefa CryptoManager é responsável por criar um conjunto de dados aleatórios para ser criptografado e descriptografado a cada meio segundo. O tamanho deste conjunto de dados, associado com o período de tempo de meio segundo, simula um canal de comunicação serial de taxa fixa. Por exemplo, um conjunto de dados de 8.192 bytes gerado a cada meio segundo simula um canal de comunicação serial *full duplex* de 128Kbps. Ou seja, 8.192 bytes seriam transmitidos e recebidos a cada meio segundo, e a operação de criptografia seria feita em ambos os lados da comunicação pelo programa. Nesta aplicação de teste, como o canal serial não existe fisicamente, a criptografia é feita sobre dados aleatórios, e o resultado da operação inversa da criptografia é comparado com os dados originais para verificar se houve sucesso. Com este experimento é possível saber quanto tempo um processador MPC850, operando a 48MHz e com a configuração de memória do módulo e850Lite, utilizaria para tratar em *software* a criptografia de tal canal de comunicação. As outras duas tarefas do sistema, BlowfishEncrypter e BlowfishDecrypter, são utilizadas para realizar as operações de criptografia direta e inversa, respectivamente, recebendo os dados de trabalho da tarefa CryptoManager. Apesar de ser possível fazer estas operações dentro da própria tarefa CryptoManager, a separação das operações em duas tarefas distintas permite medir mais facilmente o tempo consumido por cada uma das etapas, separando também o tempo de gerenciamento e emulação do canal serial dos tempos utilizados no algoritmo de criptografia.

7.3 DESEMPENHO DO SISTEMA OPERACIONAL

Uma questão relacionada ao desenvolvimento de sistemas operacionais tempo real é a avaliação de desempenho do núcleo de tempo real e o diagnóstico de eventuais problemas relativos ao tempo de execução dos seus serviços. Para verificar de que modo o Dyretiva e o SoftScope podem auxiliar em tais tarefas, foram feitos experimentos com o PET# no ambiente de teste. O núcleo do PET# é formado por serviços de troca de mensagens que executam muitas vezes por segundo. Problemas temporais nestes serviços poderiam provocar atrasos na aplicação, com possíveis perdas de prazo. O cenário criado para este teste configurou o canal de comunicação serial emulado para operar em taxas variando de zero a 512Kbps por um período de aproximadamente dois minutos. Como os dados foram obtidos em diferentes arquivos de rastreamento, uma compilação destas informações é apresentada na Tabela 11.

Tabela 11: Tempos de execução dos serviços de trocas de mensagens do PET#.

Serviço	Execuções	OBCET	OTCET	OWCET
PutMessage	6.680	13,4µs	17,8µs	19,7µs
SendMessage	6.834	17,4µs	22,6µs	25,6µs
ReceiveMessage (vazia)	13.878	9,9µs	12,8µs	21,0µs
ReceiveMessage (Put)	13.077	11,8µs	13,3µs	15,6µs
ReceiveMessage (sem nova)	5.998	16,7µs	17,0µs	17,7µs
ReceiveMessage (SendMessage)	6.834	13,3µs	15,2µs	22,3µs
ReplyMessage	6.835	14,2µs	17,0µs	19,2µs

A primeira coluna da Tabela 11 mostra os nomes dos serviços de troca de mensagens no PET#. O PutMessage é o serviço que envia uma mensagem assíncrona de uma tarefa para outra, enquanto o SendMessage é o serviço que envia uma mensagem síncrona. O ReceiveMessage é utilizado por uma tarefa para receber uma mensagem, tanto assíncrona como síncrona. Para facilitar a análise, o ReceiveMessage foi dividido em quatro casos distintos. O primeiro caso refere-se a medições feitas quando o ReceiveMessage é chamado e não há mensagens disponíveis. O segundo refere-se a medições feitas quando o ReceiveMessage é chamado e encontra uma mensagem assíncrona disponível. As mensagens assíncronas podem ser enviadas para uma tarefa por outra tarefa, utilizando o serviço de PutMessage, ou por rotinas de atendimento a interrupções, utilizando o serviço de PutMessageISR. O terceiro caso refere-se a medições feitas quando o ReceiveMessage é chamado e não há mensagens novas na fila, ou seja, há apenas mensagens já recebidas, mas

ainda não respondidas. Por fim, o último caso refere-se a medições feitas quando o ReceiveMessage encontra uma mensagem síncrona disponível. Logo após os quatro casos do ReceiveMessage, a tabela apresenta o ReplyMessage, que é o serviço utilizado para responder a uma mensagem síncrona.

Seguindo na Tabela 11, a segunda coluna mostra o número de execuções de cada serviço. A quarta coluna mostra o OBCET (Observed Best Case Execution Time), que é o menor valor entre todas as amostras analisadas. A quinta coluna mostra o OTCET (Observed Typical Case Execution Time), que é a média aritmética de todas as amostras analisadas. A sexta e última coluna mostra o OWCET (Observed Worst Case Execution Time), que é o maior valor de todas as amostras.

Com os dados apresentados na Tabela 11 constata-se que os serviços de troca de mensagens no PET# possuem tempos de execução variando entre 9,9µs, no melhor caso, e 25,6µs, no pior caso. Se considerarmos todos os serviços em conjunto, o tempo médio de execução é de cerca de 16,5µs, ou seja, cerca de 800 ciclos de relógio em um processador de 48MHz. Este tempo médio inclui a operação de salvamento de contexto da tarefa solicitante, a execução do serviço solicitado, a determinação da próxima tarefa a executar e a recuperação do contexto desta próxima tarefa.

Apesar do tempo médio de execução de cada serviço ser regular, a Tabela 11 mostra que em alguns casos o tempo de execução observado no pior caso foi muito maior que o tempo médio. Este é o caso da terceira linha da tabela, quando uma tentativa de recebimento de mensagem sem que uma esteja disponível utilizou 12,8µs em média, mas demorou 21µs no pior caso observado. Este fenômeno também se repetiu na penúltima linha da tabela, quando o tempo médio observado para o recebimento de uma mensagem síncrona foi de 15,2µs, mas o pior caso mediu 22,3µs. Utilizando o SoftScope foi possível identificar a causa deste comportamento, que refere-se a ocorrência de uma interrupção de relógio durante a execução da chamada de sistema ReceiveMessage. Isto ocorre porque o PET# permite que os serviços do núcleo sejam interrompidos por interrupções de *hardware*, com vistas a diminuir a latência no atendimento de interrupções. No entanto, nestes casos o tempo de atendimento à interrupção do relógio acaba sendo computado na duração da chamada de sistema, pois do ponto de vista do usuário que fez a chamada esta é a duração real do seu pedido. Com isto, o comportamento resultante dá a impressão errônea de que há algo errado com a implementação do serviço ReceiveMessage. É importante notar que, apesar deste comportamento ter sido observado apenas durante a execução do serviço ReceiveMessage, todas as outras chamadas de sistema também são susceptíveis a este comportamento.

Em resumo, nesta seção foi mostrado como o SoftScope e o Dyretiva podem ser usados para analisar o desempenho temporal de um sistema operacional tempo real, bem como para diagnosticar eventuais problemas relacionados com o tempo de execução dos seus serviços.

7.4 TAXA DE UTILIZAÇÃO DO PROCESSADOR

A taxa de utilização do processador é fornecida pelo SoftScope na fase de filtragem e análise dos dados (Seção 6.6) como saída do analisador do rastro, na forma de estatísticas da execução. Posteriormente, esta mesma informação é apresentada em tabelas e gráficos de setores pelo programa de apresentação dos resultados (Seção 6.7). A análise de resultados gerados pelo Dyretiva e pelo SoftScope nesta seção refere-se a um cenário de teste no qual a IUT foi configurada para simular o canal de comunicação serial a uma taxa de 256Kbps por um período de cerca de 20 segundos. Com os dados coletados neste experimento foi feita a medição e a análise do tempo de utilização do processador, conforme mostrado na Tabela 12 e que é a mesma informação contida no gráfico superior da Figura 39.

Tabela 12: Tempo de utilização do processador.

Descrição	Tempo	Tempo (%)
Tempo ocioso	14,4s	71,76%
Tempo de sistema (execução dos serviços de troca de mensagem no núcleo)	238,3ms	1,18%
Tempo utilizado pelas tarefas	5,4s	27,06%
Tempo total	20,1s	100%

A Tabela 12 está organizada de forma que as três primeiras linhas contêm o tempo ocioso, o tempo de sistema e o tempo das tarefas respectivamente. A segunda coluna apresenta o tempo absoluto utilizado por cada item durante a amostragem, enquanto a terceira coluna mostra este mesmo valor em termos percentuais. A última linha apresenta a totalização do teste, mostrando que foram monitorados 20,1 segundos de execução, o que corresponde a 100% de tempo do processador.

Numa análise dos dados da Tabela 12 pode-se destacar o tempo gasto com chamadas de sistema foi pequeno, de pouco mais de um por cento. Este valor representa todos os serviços de troca de mensagens do PET#. Já o tempo gasto na execução das tarefas foi de cerca de vinte e sete por cento do total. Para analisar esta informação em particular, este valor foi dividido entre as tarefas ativas, conforme apresentado na Tabela 13 e que é a mesma informação contida no gráfico inferior da Figura 39.

Tabela 13: Utilização do processador por tarefa.

Tarefa	Prioridade	Preempções	Tempo de CPU	Tempo de CPU (%)
TimeManager	0	0	71,2ms	0,35%
ttyManager	7	0	64,9ms	0,32%
Hearbeat(GREEN)	16	0	1,3ms	0,01%
Heartbeat(RED)	25	0	1,2ms	0,01%
CryptoManager	22	40	235ms	1,17%
BlowfishEncrypter	21	120	2,5s	12,58%
BlowfishDecrypter	21	120	2,5s	12,62%

A primeira coluna da Tabela 13 mostra o nome das tarefas que executaram durante a monitoração. A segunda coluna mostra, por questões de referência, a prioridade de cada tarefa, sendo que no PET# quanto menor o número maior a prioridade da tarefa. A terceira coluna informa quantas vezes cada tarefa sofreu preempção. A quinta coluna mostra o tempo total utilizado por cada tarefa e a sexta e última colunas mostram este mesmo tempo em termos percentuais aos 20,1 segundos do teste. A soma de todos os percentuais da última coluna deve ser de 27,06%, o que corresponde à penúltima linha da Tabela 12.

A duas primeiras linhas da Tabela 13 apresentam o tempo utilizado pelas tarefas TimeManager e ttyManager. Estas tarefas, apesar de executarem no espaço do usuário, implementam funcionalidades do sistema operacional e possuem tempos de execução pequenos. As duas tarefas seguintes, Heartbeat(GREEN) e Heartbeat(RED), fazem parte do *software* que suporta os dispositivos encontrados no módulo e850Lite, e seu tempo de execução também não é significativo. Finalmente, as três última tarefas, CryptoManager, BlowfishEncrypter e BlowfishDecrypter, formam a aplicação sob teste e utilizam juntas a maior parte do tempo não ocioso do processador. Analisando a aplicação sob teste, nota-se que pouco mais de um quarto do tempo do processador está sendo utilizado apenas pelo algoritmo de criptografia sobre um canal serial de 256Kbps, sendo cerca de um oitavo do tempo total para criptografar os dados a serem enviados, e outro um oitavo para descriptografar os dados recebidos. Considerando que velocidades de 256Kbps não são muito elevadas para canais seriais, e que o algoritmo de criptografia utilizou um quarto do tempo do processador, o experimento demonstra o motivo pelo qual muitos processadores implementam unidades de *hardware* dedicadas para tratar algoritmos de criptografia.

7.5 TEMPO DE EXECUÇÃO DE FUNÇÕES

O Dyretiva e o SoftScope também podem ser utilizados na medição do tempo de execução de funções. Para exemplificar, na IUT configurada para trabalhar com o canal serial a uma taxa de 256Kbps, três funções periódicas foram monitoradas: a função de tratamento dos terminal TTY e as funções que realizam operações de criptografia Blowfish em dados. O resultado apurado pelo SoftScope é mostrado na Figura 47.

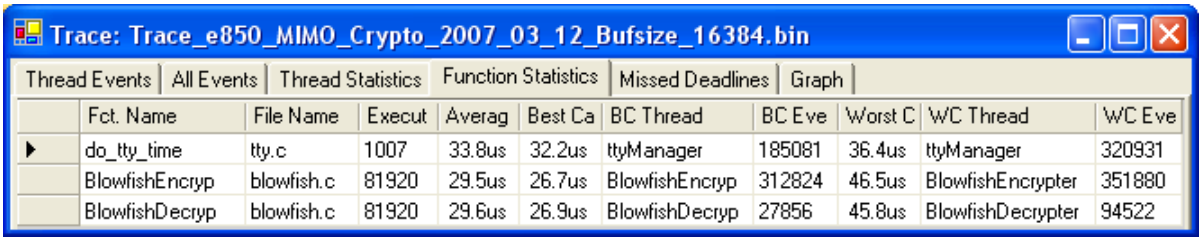


Figura 47: Tempos de execução de funções monitoradas.

Para cada função, o SoftScope mostra seu nome na primeira coluna, o arquivo fonte onde está escrita na segunda coluna e o número de execuções observadas na terceira coluna. As 1007 execuções da função `do_tty_time` são um valor esperado, pois o rastro possui 20,1 segundos de tempo total de execução e o período da tarefa é de 20ms. As funções `BlowfishEncryp` e `BlowfishDecryp` foram executadas muito mais vezes, em virtude de serem chamadas a cada bloco de 8 bytes a ser criptografado ou descriptografado. A quarta, quinta e sétima colunas trazem o OTCET, o OBCET e o OWCET, respectivamente. Com estes dados pode-se afirmar que o tempo típico é bastante próximo ao tempo médio, mas os tempos no piores casos são quase o dobro dos nos melhores casos. Para verificar o que ocorre em um dos piores casos, o comportamento do `BlowfishEncryp` a partir do evento número 351.880 foi apurado e é mostrado na Figura 48. O número do evento no arquivo de rastreamento que aponta para o pior caso de tempo de execução das funções é mostrado na última coluna da tabela de tempos de execução na Figura 47.

Trace: Trace_e850_MIMO_Crypto_2007_03_12_Bufsize_16384.bin

Thread Events | All Events | Thread Statistics | Function Statistics | Missed Deadlines | Graph

89%	Event Nr	Time Sta	Event	Event Name	Nr	Param 1	Param 2	Par	Para
	351880	1280	USER	USER	1	1	0	0	0
	351881	13480	PET#	PET_EXCEPTION_ENT	1	Decrementer	0	0	0
	351882	9280	PET#	PET_PUT_MSG_ISR	1	TimeManager	0	0	0
	351883	10800	PET#	PET_SCHEDULE	1	TimeManager	0	0	0
	351884	1280	PET#	PET_EXCEPTION_EXIT	0	0	0	0	0
	351885	8120	PET#	PET_EXCEPTION_ENT	1	SystemCall	0	0	0
	351886	6020	PET#	PET_RECEIVE_PUT_M	1	TimeManager	0	0	0
	351887	7260	PET#	PET_EXCEPTION_EXIT	0	0	0	0	0
	351888	6180	PET#	PET_TIMER_EXPIRED	1	ttyManager	0	0	0
	351889	5120	PET#	PET_EXCEPTION_ENT	1	SystemCall	0	0	0
	351890	7500	PET#	PET_PUT_MSG	2	TimeManager	ttyManager	0	0
	351891	10360	PET#	PET_EXCEPTION_EXIT	0	0	0	0	0
	351892	11480	PET#	PET_EXCEPTION_ENT	1	SystemCall	0	0	0
	351893	5300	PET#	PET_RECEIVE_MSG_E	1	TimeManager	0	0	0
	351894	5720	PET#	PET_SCHEDULE	1	ttyManager	0	0	0
	351895	2020	PET#	PET_EXCEPTION_EXIT	0	0	0	0	0
	351896	7820	PET#	PET_EXCEPTION_ENT	1	SystemCall	0	0	0
	351897	7260	PET#	PET_RECEIVE_PUT_M	1	ttyManager	0	0	0
	351898	6160	PET#	PET_EXCEPTION_EXIT	0	0	0	0	0
	351899	4200	USER	USER	1	5	0	0	0
	351900	34580	USER	USER	1	6	0	0	0
	351901	9520	PET#	PET_EXCEPTION_ENT	1	SystemCall	0	0	0
	351902	7900	PET#	PET_SEND_MSG	2	ttyManager	TimeManager	0	0
	351903	14200	PET#	PET_SCHEDULE	1	TimeManager	0	0	0
	351904	1060	PET#	PET_EXCEPTION_EXIT	0	0	0	0	0
	351905	9200	PET#	PET_EXCEPTION_ENT	1	SystemCall	0	0	0
	351906	9600	PET#	PET_RECEIVE_SEND_	2	TimeManager	ttyManager	0	0
	351907	5620	PET#	PET_EXCEPTION_EXIT	0	0	0	0	0
	351908	6560	PET#	PET_SET_TIMER	2	ttyManager	20	0	0
	351909	10600	PET#	PET_EXCEPTION_ENT	1	SystemCall	0	0	0
	351910	10240	PET#	PET_REPLY_MSG	2	TimeManager	ttyManager	0	0
	351911	7220	PET#	PET_EXCEPTION_EXIT	0	0	0	0	0
	351912	6600	PET#	PET_EXCEPTION_ENT	1	SystemCall	0	0	0
	351913	5480	PET#	PET_RECEIVE_MSG_E	1	TimeManager	0	0	0
	351914	5220	PET#	PET_SCHEDULE	1	ttyManager	0	0	0
	351915	2220	PET#	PET_EXCEPTION_EXIT	0	0	0	0	0
	351916	8800	PET#	PET_EXCEPTION_ENT	1	SystemCall	0	0	0
	351917	8800	PET#	PET_RECEIVE_MSG_N	1	ttyManager	0	0	0
	351918	6520	PET#	PET_SCHEDULE	1	BlowfishEncr	0	0	0
	351919	1440	PET#	PET_EXCEPTION_EXIT	0	0	0	0	0
	351920	33060	USER	USER	1	2	0	0	0

Figura 48: Eventos no pior caso de tempo de execução da função BlowfishEncryp.

Na Figura 48 observa-se que entre o evento número 351.880 – que é o evento número um, ligado à instrumentação do usuário, e que marca o início da execução da função `BlowfishEncryp` – e o evento 351.920 – que é o evento número dois, também ligado à instrumentação do usuário, e que marca o final da execução da função – houve preempção da tarefa que executava a função. Como era esperado, o SoftScope computou corretamente o tempo da função, que é composto da soma do tempo relacionado ao evento 351.881, no valor de 13,5µs, correspondente ao tempo decorrido entre o início da função `BlowfishEncryp` e o início de uma exceção, com o tempo relacionado ao evento 351.920, no valor de 33,1µs, correspondente ao período entre o final de uma exceção tratada pelo sistema operacional e o final da execução da função `BlowfishEncryp`. Os outros tempos envolvidos não são computados no tempo de execução da função, já que pertencem ao sistema ou a outras tarefas. A Figura 48 explica o motivo que levou este a ser o pior caso do tempo de execução, e que está ligado à preempção da função em meio à execução e não à implementação do algoritmo da função.

7.6 VERIFICAÇÃO DAS RESTRIÇÕES TEMPORAIS

Para verificar a eficácia do Dyretiva e do SoftScope em detectar violações nas restrições temporais, foi criada uma restrição temporal de acordo com o modelo de falta do Dyretiva para a aplicação de teste. Neste experimento, o canal de comunicação serial foi simulado a uma taxa de 512Kbps, com as três tarefas de gerenciamento e execução da criptografia sendo consideradas a restrição temporal. O período estabelecido para a restrição temporal foi 500ms, enquanto o prazo estabelecido foi de 259ms. O arquivo que contém a descrição da restrição temporal utilizada neste caso é apresentado na Figura 49.

```
[NUMBER_OF_TIME_REQUIREMENTS = 1]
[TIME_REQUIREMENT_NAME = Blowfish]
[START_EVENT = OS, 2, 82, 9, 10]
[END_EVENT = OS, 2, 87, 11, 9]
[RELATED_THREADS = 3]
[THREAD1 = CryptoManager]
[THREAD2 = BlowfishEncrypter]
[THREAD3 = BlowfishDecrypter]
[DEADLINE = 259]
[INTERVAL = 500]
```

Figura 49: Conteúdo do arquivo de restrições temporais.

Na Figura 49 a restrição temporal é chamada de Blowfish, o evento inicial é o pedido de criptografia da tarefa CryptoManager para a tarefa BlowfishEncrypter, enquanto o evento final é uma resposta da tarefa BlowfishDecrypter para a tarefa CryptoManager, indicando o fim da criptografia.

Ao aplicar as restrições temporais ao arquivo de rastreamento obtido com o MIMO, o resultado da análise temporal da restrição Blowfish é mostrado na Figura 50 e indica duas violações ao prazo estabelecido de 259ms. A primeira violação é reportada em detalhes pelo relatório gerado. Além disso, o tempo de execução no melhor e no pior caso em que o prazo é cumprido são indicados na parte inferior da janela. Para visualizar em que circunstâncias a violação temporal ocorreu, um gráfico de Gannt do trecho onde a falta de prazo foi encontrada pode ser criado pelo SoftScope, como o que é mostrado na Figura 51.

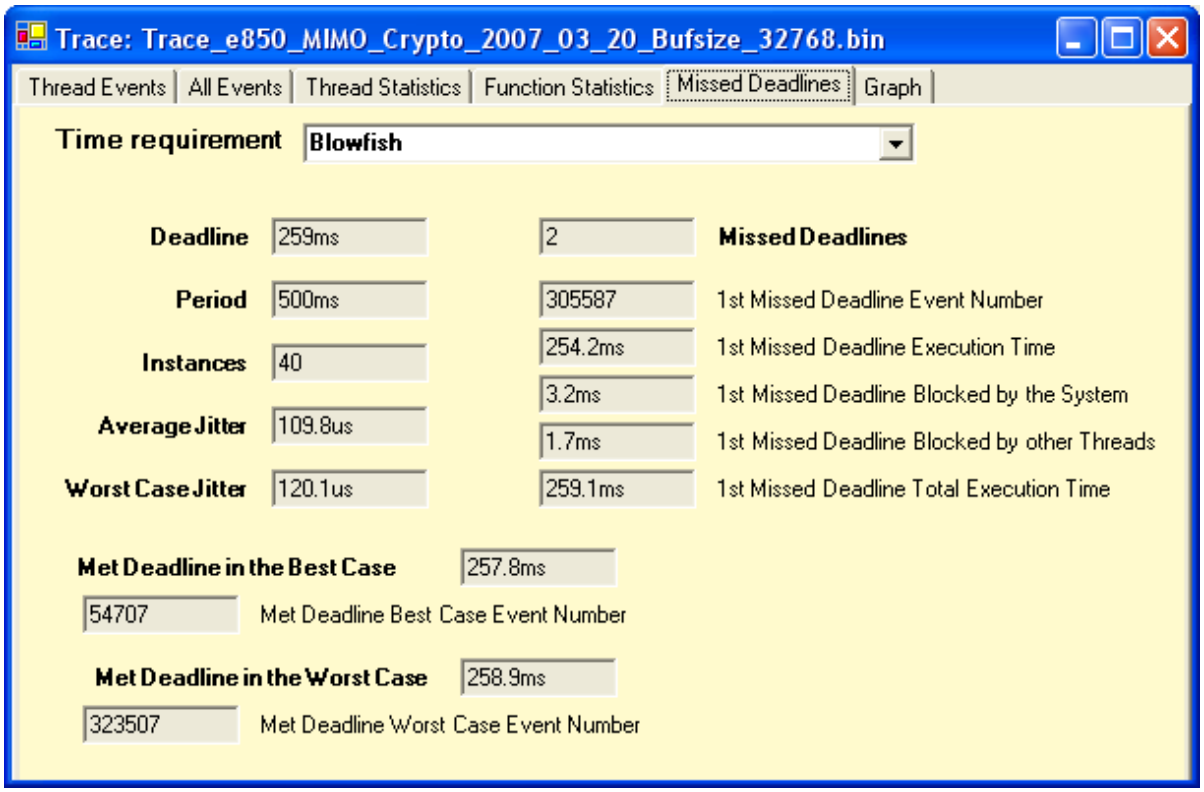


Figura 50: Análise de prazo para a restrição temporal de teste.

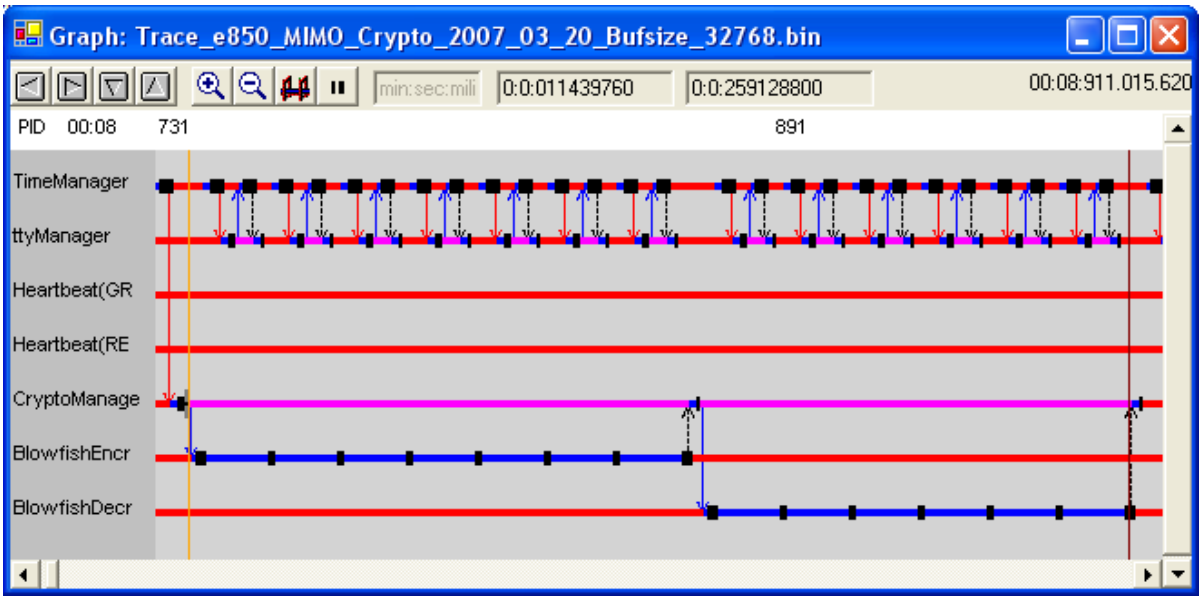


Figura 51: Comportamento dinâmico da violação temporal detectada.

Na Figura 51 não se percebe nenhum tipo de comportamento anormal das tarefas envolvidas com a restrição temporal, além da esperada preempção pelas tarefas de mais alta prioridade do sistema. Na verdade, neste exemplo o prazo foi reduzido propositalmente de modo a forçar uma violação, permitindo uma ilustração de como a ferramenta calcula e reporta as violações e, também, a forma com que o usuário pode utilizar o SoftScope para identificar um problema temporal.

7.7 INTRUSÃO DA MONITORAÇÃO HÍBRIDA

Os dados de rastreamento utilizados nos experimentos deste capítulo foram coletados pelo MIMO, o monitor híbrido do SoftScope. Entretanto, uma questão importante a ser analisada é a intrusão causada pela instrumentação que permite que o MIMO colete estes dados, inclusive porque o Dyretiva propõe que esta instrumentação seja feita permanente para evitar o efeito sonda.

Para medir este valor foi utilizada uma IUT similar àquela descrita na Seção 7.2, porém sem as três tarefas da aplicação (CryptoManager, BlowfishEncrypter e BlowfishDecrypter), que foram substituídas por um único comando no contexto do interpretador do PET#. Este comando marca o instante de seu início e executa, com baixa prioridade, uma operação de criptografia direta e reversa em uma quantidade de dados fixa de 2MB. Como esta operação é feita em baixa prioridade, ela sofre a influência de todo o restante do sistema operacional e das tarefas de suporte ao módulo e850Lite. Ao final da

execução o tempo é anotado novamente e, quando subtraído do tempo de início, resulta o tempo necessário para executar esta quantidade fixa de trabalho.

A avaliação da intrusão causada pela instrumentação é feita utilizando duas versões da aplicação de teste: uma sem instrumentação alguma e outra com instrumentação. Na versão não instrumentada, nem o sistema operacional e nem a aplicação possuem qualquer instrução de instrumentação. Na versão instrumentada tanto o sistema operacional quanto a aplicação possuem instrumentação. A instrumentação da aplicação foi inserida na entrada e na saída das funções que realizam a criptografia Blowfish. Estas funções são chamadas a cada 64 bits de dados a serem criptografados ou descriptografados. Isto significa que a operação completa de criptografia em 2MB de dados implica 524.288 chamadas a cada uma das duas funções, ou 262.144 chamadas para cada uma delas. Como cada função é instrumentada em dois pontos, que são a entrada e a saída, somente a aplicação gera 1.048.576 chamadas à função de instrumentação que escreve na porta de medição. Além disso, como o sistema operacional também está instrumentado, a influência da instrumentação do sistema operacional também é considerada pelo experimento. Os resultados obtidos neste experimento são mostrados na Tabela 14.

Tabela 14: Medida da intrusão causada pela monitoração híbrida.

Situação	Média do Tempo de Execução
Sem Instrumentação	52,504038867 segundos
Com Instrumentação	53,564690734 segundos
Diferença :	1,98%

Na primeira linha da Tabela 14 encontra-se o tempo médio de execução da IUT sem instrumentação, que é de cerca de cinqüenta e dois segundos e meio. Já na linha seguinte encontra-se o tempo médio de execução da IUT com instrumentação, tanto no sistema operacional quanto na aplicação, que é de cerca de cinqüenta e três segundos e meio. Considerando que somente a aplicação gera 1.048.576 de chamadas à função de instrumentação, pode-se concluir que na versão instrumentada do programa uma invocação à função de instrumentação é feita, no mínimo, a cada 50µs. Esta conta não leva em consideração que o sistema operacional também está instrumentado. Nestas circunstâncias, a última linha da tabela mostra que a sobrecarga causada pela instrumentação é de cerca de dois por cento. Considerando o objetivo do Dyretiva em não remover a instrumentação do programa, este nível de intrusão pode ser considerado aceitável.

7.8 RESUMO

Neste capítulo foram apresentadas situações onde o Dyretiva e o SoftScope foram utilizados com o objetivo de obter informações sobre o comportamento dinâmico e a correção temporal de um sistema embarcado real. Estas situações foram baseadas em uma aplicação de teste que simulou uma atividade com características de tempo real. Os dados de rastreamento foram coletados com o monitor híbrido MIMO.

As situações exercitadas mostraram que o método captura as informações necessárias para realizar a verificação temporal de um ERTS, sendo capaz de identificar faltas de concorrência, faltas de processamento e faltas de prazo. Entre as situações exercitadas pelos casos de teste propostos estão a avaliação de desempenho do sistema operacional, a avaliação de desempenho da aplicação, a avaliação temporal de algoritmos, e a busca de violações temporais no sistema a partir do conceito de restrição temporal do Dyretiva (Seção 5.4.3).

Além disso, a partir de um exemplo que utilizou instrumentação intensivamente, foi medida a intrusão causada pela abordagem de instrumentação do Dyretiva no sistema monitorado. Com a obtenção de um valor de menos de dois por cento neste quesito, é possível concluir que a coleta de dados está sendo feita com baixa intrusão, e que a permanência das instruções de instrumentação no *software* com o objetivo de eliminar o efeito sonda é viável.

No próximo capítulo são apresentadas as conclusões do trabalho, que incluem as contribuições, as publicações e as sugestões para trabalhos futuros.

8 CONCLUSÃO

Sistemas embarcados operando em tempo real podem ser encontrados em diversas áreas de aplicação, desde dispositivos eletro-eletrônicos de uso doméstico até sistemas digitais de controle de plantas nucleares. Uma característica comum a todos eles é a existência de uma especificação temporal, que exige que os resultados sejam produzidos dentro de limites de tempo bem definidos. Além disso, se espera que os ERTS sejam previsíveis, confiáveis, robustos e seguros, características que dependem basicamente da qualidade do *software* embarcado e que podem ser avaliadas efetivamente durante a fase de testes do sistema.

Em virtude das características intrínsecas dos ERTS, o desenvolvimento de *software* para tais sistemas apresenta desafios peculiares, como a necessidade de realizar uma verificação temporal em sistemas concorrentes, dedicados e que possuem recursos limitados. Como esta não é uma realidade nos sistemas de computação com o objetivo de processamento de dados, a verificação temporal exige métodos e ferramentas desenvolvidos especialmente para a tarefa a que se destinam.

Um estudo de métodos e ferramentas voltados para a verificação temporal (Seção 4.4) revelou que as necessidades de testes dos ERTS não são plenamente atendidas por eles. Em função destas lacunas foram definidos os requisitos do método Dyretiva (Seção 5.1), que são: independência da arquitetura do processador utilizado; monitoração híbrida do SUT com imunidade ao efeito sonda; instrumentação automática e seletiva da aplicação; independência do sistema operacional; instrumentação eficiente do sistema operacional; associação com um conjunto de ferramentas que auxiliem na aplicação do método.

Em função dos requisitos, o método Dyretiva é definido em duas partes: a abordagem de monitoração e o modelo de falta. A abordagem de monitoração (Seção 5.3) define a interface física e a interface lógica entre o SUT e o monitor, a forma de instrumentar a aplicação e o sistema operacional, e o mecanismo de coleta dos dados de monitoração. É premissa da abordagem de monitoração uma plataforma de *hardware* que possua recursos limitados. O objetivo da abordagem de monitoração é criar uma visão consistente do SUT para as ferramentas de verificação temporal, independente do processador da plataforma alvo ou das ferramentas utilizadas no desenvolvimento do *software* embarcado.

A outra parte do Dyretiva é o modelo de falta (Seção 5.4), que define as relações e componentes do SUT onde se presume haver maior probabilidade de encontrar faltas. O modelo de falta do Dyretiva reflete três problemas comuns encontrados em ERTS: concorrência, processamento e perda de prazo. As faltas de concorrência podem ser causadas por preempções, interrupções ou sincronização. As faltas de processamento acontecem porque algoritmos ou tarefas utilizam o processador por mais tempo do que o esperado. Já as faltas de prazo são a maneira com que o Dyretiva detecta violações da especificação temporal.

Para auxiliar na aplicação do método Dyretiva foi desenvolvido um conjunto de ferramentas chamado de SoftScope, que é composto por um pré-instrumentador de código fonte C (Seção 6.1), um instrumentador de código fonte C (Seção 6.2), um monitor híbrido externo (Seção 6.3), um programa de acesso ao monitor (Seção 6.5), programas de filtragem e análise dos dados coletados (Seção 6.6), e um programa de apresentação dos resultados (Seção 6.7). O pré-instrumentador e o instrumentador de código fonte são utilizados no processo de instrumentação do código fonte do usuário (Seção 5.3.6). O monitor híbrido coleta os eventos gerados pelo SUT, realiza a marcação de tempo e comunica-se com o usuário (Seção 5.3.5). O programa de acesso ao monitor permite que o usuário acesse, comande a operação e leia os dados de rastreamento do monitor. Os programas de filtragem e análise dos dados têm a finalidade de obter informações a partir dos dados de rastreamento, o que é feito em três fases: decodificação, análise e avaliação dos eventos. Tecnicamente as informações geradas pelos programas de filtragem e análise dos dados são o resultado da verificação temporal do SUT, mas para facilitar a interpretação dos resultados o SoftScope também disponibiliza uma ferramenta gráfica capaz de apresentar os resultados de quatro formas diferentes: janelas informativas, gráficos de setores, tabelas e gráficos e Gannt.

A validação do Dyretiva e do SoftScope foi feita com experimentos que utilizaram o módulo e850Lite como plataforma de *hardware*, o PET# como sistema operacional, e uma aplicação de teste com características semelhantes às de um sistema em tempo real. Para os casos de teste propostos, o Dyretiva e o SoftScope foram utilizados para explicar o comportamento dinâmico e temporal do sistema. Os experimentos avaliaram o desempenho do sistema operacional, a utilização do processador, o tempo de execução de funções, e o cumprimento das restrições temporais. Por fim, foi medida também a intrusão causada pela abordagem de monitoração do Dyretiva no SUT, encontrando-se um valor abaixo de dois por cento na configuração de teste proposta.

Com base nos resultados obtidos nos experimentos de validação do método é possível afirmar que o objetivo de analisar dinamicamente o sistema com vistas a verificar as restrições temporais de um ERTS foi alcançado pelo Dyretiva e pelo SoftScope, contribuindo para aumentar a confiabilidade do sistema através da melhoraria da qualidade do *software* embarcado.

8.1 CONTRIBUIÇÕES

A principal contribuição do Dyretiva é a definição de um método de monitoração da dinâmica do sistema para utilização durante a fase de testes de ERTS, e em especial na avaliação das restrições temporais. Esta contribuição é importante porque focaliza o problema do ponto de vista das dificuldades encontradas pelas equipes de desenvolvimento, e que não eram abordadas pelos métodos e ferramentas pesquisados. O Dyretiva assume que a verificação temporal é uma etapa importante na garantia da qualidade de *software* do ERTS, e que o sistema observado é concorrente, possui recursos limitados e precisa atingir um alto grau de confiabilidade. Além disso, foram consideradas questões de ordem prática, tais como a dificuldade de migrar de uma família de processadores para outra, ou de um sistema operacional embarcado para outro, sem perder o investimento e o treinamento no uso de ferramentas de teste e de verificação temporal. Como resultado, o Dyretiva e o SoftScope foram construídos de forma a serem independentes de compilador, do ambiente de desenvolvimento, da arquitetura do processador e do sistema operacional utilizado. Além disso, a tecnologia utilizada para a monitoração do SUT é aberta, permitindo modificações e extensões, conforme necessário.

Um problema encontrado na monitoração de ERTS é a ocorrência de falhas intermitentes, que podem ser colocadas no sistema em virtude da inserção ou da remoção de rotinas de observação, fenômeno este conhecido como efeito sonda. Para eliminá-lo, o Dyretiva tomou como premissa que a intrusão no sistema monitorado deve ser minimizada, tanto pela intrusão no *software* monitorado, quanto pela seletividade com que as instruções de instrumentação podem ser colocadas na aplicação pelo usuário. Com esta minimização, presumiu-se ser possível manter a instrumentação no *software* embarcado, eliminando por completo o efeito sonda e ganhando ainda a possibilidade de observar o sistema em qualquer fase do ciclo de vida do produto. Com base em experimentos práticos comprovou-se que tal abordagem é viável, obtendo-se uma intrusão de cerca de dois por cento para a situação de

teste criada. Com isto, é possível afirmar que o Dyretiva pode ser utilizado para testar ERTS de segurança crítica sem interferir significativamente no sistema monitorado.

8.2 PUBLICAÇÕES

A pesquisa relacionada ao Dyretiva gerou três publicações em conferências nacionais e internacionais. A primeira está relacionada ao *hardware* do módulo e850Lite [Pinho 03], e descreve o desenvolvimento desta plataforma com ênfase na compatibilidade eletromagnética. A segunda refere-se ao SoftScope [Cadamuro 06], que é o conjunto de ferramentas que permite a aplicação do método Dyretiva. A terceira publicação está relacionada com a implementação do monitor híbrido MIMO [Copetti 07]. Além das publicações, submissões a outros congressos e conferências internacionais trouxeram sugestões que auxiliaram no direcionamento do projeto de pesquisa.

8.3 SUGESTÕES PARA TRABALHOS FUTUROS

Durante a definição do Dyretiva e o desenvolvimento do SoftScope foram identificadas questões que podem ser tratadas em trabalhos futuros, dentre os quais podemos destacar os itens listados abaixo.

- Incorporar os testes de cobertura e as formas de visualização do TLM ao SoftScope, gerando uma ferramenta única e com maior abrangência para uso durante a fase de testes de ERTS.
- Estender o instrumentador de código fonte do SoftScope para trabalhar com a linguagem C++, que também é bastante popular na programação de ERTS, porém mais complexa.
- Estender as ferramentas de filtragem e análise dos dados capturados para uso com outros sistemas operacionais embarcados, além do PET#.
- Gerar casos de teste temporais em função das estimativas de tempo geradas pelos métodos estáticos existentes no PERF.
- Realimentar a análise de escalonamento do PERF com os tempos das tarefas, obtidos da monitoração do SUT, permitindo análises iterativas do *software*.
- Propor e desenvolver um mecanismo que utilize os dados gerados pela monitoração para realimentar os modelos de estimação temporal e de análise de escalonamento do PERF, aperfeiçoando as estimativas e permitindo modelagens mais precisas.

Estas sugestões foram formuladas no intuito de contribuir para a melhoria contínua da usabilidade e dos resultados gerados pelo PERF, permitindo que o objetivo de construir um ambiente completo de verificação de ERTS seja alcançado no futuro.

REFERÊNCIAS

[Agilent 05] *Application Support for Agilent Logic Analyzers*. Guia de Configuração de Produtos Agilent. Documento 5966-4365E. Agilent Technologies, Inc., E.U.A., Maio/2005.

[Apvrille 04] Apvrille, L; Courtiat, J.; Lohr, C.; Saqui-Sannes, P. – *TURTLE: A Real-Time UML Profile Supported by a Formal Validation Toolkit*. Em IEEE Transactions on Software Engineering. Pág. 473-487, v. 30, n. 7, Julho/2004. IEEE Computer Society.

[Bellini 00] Bellini, P.; Mattolini, R.; Nesi, P. – *Temporal Logics for Real-Time System Specification*. Em ACM Computer Surveys. Pág. 31, Dezembro/2000. Association for Computing Machinery.

[Bellini 01] Bellini, P.; Nesi, P. – *TILCO-X, an Extension of TILCO Temporal Logic*. Em 7th IEEE International Conference on Engineering of Complex Computer Systems (ICECCS'01). Pág. 0015, Junho/2001. IEEE Computer Society.

[Bellini 06] Bellini, P.; Nesi, P.; Rogai, D. – *Reply to Comments on "An Interval Logic for Real-Time System Specification"*. Em IEEE Transactions on Software Engineering. Pág. 428-431, v. 32, n. 6, Junho/2006. IEEE Computer Society.

[Binder 01] Binder, R. – *Testing Object-Oriented Systems: Models, Patterns, and Tools*. Editora Addison-Wesley. Terceira Impressão, E.U.A., 2001.

[Braga 98] Braga, A.; Renaux, D. – *Utilização de Linhas de Tempo para Depuração e Validação Temporal de Sistemas em Tempo Real*. I Workshop de Sistemas em Tempo Real, Rio de Janeiro, Maio/1998.

[Braga 99a] Braga, A. S. – *TLM – Uma Ferramenta de Apoio ao Teste de Restrições Temporais em Sistemas Dedicados Operando emTempo Real*. Dissertação de Mestrado apresentada ao Programa de Pós-Graduação em Engenharia Elétrica e Informática Industrial. CEFET-PR, Curitiba, 1999.

[Braga 99b] Braga, A. S.; Renaux, D. P. B. – *Teste de Restrições Temporais através de Técnicas Funcionais e Estruturais*. II Workshop em Sistemas de Tempo Real, Salvador, Maio/1999.

[Bregant 02] Bregant, E. G. – *Análise de Escalonamento de Tarefas no Ambiente PERF*. Dissertação de Mestrado apresentada ao Programa de Pós-Graduação em Engenharia Elétrica e Informática Industrial. CEFET-PR, Curitiba, 2002.

[Bregant 05] Bregant, E. G.; Renaux, D. P. B. – *RTOS Scheduling Analysis Using a Trace Toolkit*. Em WTR2005 -VII Workshop de Tempo Real. Fortaleza, Brasil, Maio/2005.

[Briand 99] Briand, L. P.; Roy, D. M. – *Meeting Deadlines in Hard Real-Time Systems: The Rate Monotonic Approach*. Editora IEEE Computer Society Press. ISBN 0-8186-7406-7, E.U.A., 1999.

[Burns 97] Burns, A.; Wellings, A. – *Real-Time Systems and Programming Languages*. Editora Addison-Wesley – 2ª Edição, Inglaterra, 1997.

[Cadamuro 01] Cadamuro Jr., J. – *CFPPC: Um Sistema Didático Baseado em Processadores PowerPC*. Dissertação de Mestrado apresentada ao Programa de Pós-Graduação em Engenharia Elétrica e Informática Industrial. CEFET-PR, Curitiba, 2001.

[Cadamuro 06] Cadamuro Jr., J.; Renaux, D. P. B. – *SoftScope: Embedded Real-Time Systems Verification Tool Set*. Em Proceedings of the Work in Progress Session of IEEE Real-Time Systems Symposium. Pág. 33-36, Dezembro/2006. IEEE Computer Society.

[Cargille 92] Cargille, J.; Miller, B. P. – *Binary Wrapping: A Technique for Instrumenting Object Code*. ACM SIGPLAN Notices. Pág. 17-18, v. 27, n. 6, Junho/1992. Association for Computing Machinery.

[Copetti 07] Copetti, L. F.; Cadamuro Jr., João; Renaux, D. P. B.; Pedroni, V. A. – *MIMO: A Hybrid Monitor for Embedded Real-Time Systems*. IX Workshop on Real-Time Systems (WTR 2007), Pág. 39-46, Belém, Maio/2007.

[Darlington 78] Darlington, J.; Burstall, R. – *A System which Automatically Improves Programs*. Em Programming Methodology, de David Gries. Editora Springer, Nova Iorque, 1978.

[Dees 05] Dees, R. – *Nexus Revealed: An Introduction to the IEEE-ISTO 5001 Nexus Debug Standard*. Editora AMT – Edição Preliminar, E.U.A., 2005.

[Douglass 98] Douglass, B. P. – *Real-Time UML: Developing Efficient Objects for Embedded Systems*. Editora Addison Wesley Longman, Inc.; Massachusetts, E.U.A., 1998.

[Engblom 97] Engblom, J. – *Worst-Case Execution Time Analysis for Optimized Code*. Dissertação de Mestrado apresentada ao Departamento de Sistemas da Computação da Universidade de Uppsala. Uppsala University, Suécia, 1997.

[eSysTech 03] *Manual do Usuário da Placa de Avaliação e850Lite*. Documento PR-ESYS-E850L-MAN-0100b, versão 1.00b. eSysTech Ltda., Brasil, Julho/2003.

[Gait 86] Gait, P. – *A Probe Effect in Concurrent Programs*. Software – Practice and Experience, pág. 225-233, Março/1986.

[Gebremichael 05] Gebremichael, B.; Vaandrager, F. – *Specifying Urgency in Timed I/O Automata*. Em 3rd IEEE International Conference on Software Engineering and Formal Methods (SEFM'05). Pág. 64-74, Setembro/2005. IEEE Computer Society.

[Góes 00] Góes, J. – *Estimação de Tempo de Execução de Programas a Partir de Arquivos Objeto*. Trabalho individual apresentado ao Programa de Pós-Graduação em Engenharia Elétrica e Informática Industrial. CEFET-PR, Curitiba, 2000.

[Góes 01a] Góes, J. – *PERF: Ambiente de Desenvolvimento e Estimação Temporal de Sistemas em Tempo Real*. Dissertação de Mestrado apresentada ao Programa de Pós-Graduação em Engenharia Elétrica e Informática Industrial. CEFET-PR, Curitiba, 2001.

[Góes 01b] Góes, J.; Linhares, R.; Renaux, D. – *Estimação de Tempo de Execução de Programas no Ambiente PERF*. III Workshop em Sistemas em Tempo Real. Florianópolis, Maio/2001.

[Green 07] Green Hills Software, Inc. – *MULTI Integrated Development Environment Data Sheet*. http://www.ghs.com/download/datasheets/MULTI.pdf, Agosto, 2007.

[Greenwalt 03] Greenwalt, T. – *Code Test: Logic Analyzer for Software Engineers*. Smart Networks Developer Forum 2003. Relatório da Seção Técnica K220, 35 páginas, Dallas, Texas, E.U.A., Março/2003. Motorola, Inc.

[Gupta 01] Gupta, R.; Zhang, Y. – *Timestamped Whole Program Path Representation and its Applications*. Em Proceedings of the ACM SIGPLAN 2001 Conference on Programming Language Design and Implementation. Pág. 180-190, v. 36, n. 5, Maio/2001. Association for Computing Machinery.

[Harel 87] Harel, D. – *Statecharts: A Visual Formalism for Complex Systems*. Em Science of Computer Programming 8, pág. 231-274, 1987. Elsevier Science Publishers B.V., North-Holland.

[Havelund 03] Havelund, K.; Stoller, S. D.; Ur, S. – *Bechmark and Framework for Encouraging Research on Multi-Threaded Testing Tools*. Em Proceedings of the International Parallel and Distributed Processing Symposium (IPDPS'03). Pág. 286a, Abril/2003. IEEE Computer Society.

[Healy 95] Healy, C.; Whaley, D.; Harmon, M. – *Integrating the Timing Analysis of Pipelining and Instruction Caching*. Em Proceedings of IEEE Real-Time Systems Symposium. Pág. 288, Dezembro/1995. IEEE Computer Society.

[Hennessy 96] Hennessy, J. L.; Patterson, D. A. – *Computer Architecture: A Quantitative Approach*. Editora Morgan Kaufmann – 2ª Edição, E.U.A., 1996.

[Huselius 03] Huselius, J.; Sundmark, D.; Thane, H.; – *Start Conditions for Post-Mortem Debugging using Deterministic Replay of Real-Time Systems*. Em Proceedings of the 15th Euromicro Conference on Real-Time Systems (ECRTS'03). Pág. 177, Julho/2003. IEEE Computer Society.

[Kawamura 99] Kawamura, A. – *Análise Estática de Programas para Predição de Tempos de Execução*. Dissertação de Mestrado apresentada ao Programa de Pós-Graduação em Engenharia Elétrica e Informática Industrial. CEFET-PR, Curitiba, 1999.

[Kaynar 03] Kaynar, K.; Lynch, N.; Segala, R.; Vaandrager, F. – *Timed I/O Automata: A Mathematical Framework for Modeling and Analyzing Real-Time Systems*. Em Proceedings of 24th IEEE Real-Time Systems Symposium (RTSS'03). Pág. 166, Dezembro/2003. IEEE Computer Society.

[Laplante 90] Laplante, P. – *Heisenberg Uncertainty*. Em ACM SIGSOFT Software Engineering Notes. Pág. 21-22, v. 15, n. 5, Outubro/1990. Association for Computing Machinery.

[Larus 94] Larus, J. R.; Ball, T. – *Optimally Profiling and Tracing Programs*. Em ACM Transactions on Programming Languages and Systems. Pág. 1319-1360, v. 16, n. 4, Julho/1994. Association for Computing Machinery.

[Larus 99] Larus, J. R. – *Whole Program Paths*. Em Proceedings of the ACM SIGPLAN 1999 Conference on Programming Language Design and Implementation. Pág. 259-269, v. 34, n. 5, Maio/1999. Association for Computing Machinery.

[LeBlanc 87] LeBlanc, T. J.; Mellor-Crummey, J. M. – *Debugging Parallel Programs with Instant Replay*. Em IEEE Transactions on Computers, pág. 471-482, Abril/1987.

[Linhares 01] Linhares, R. R. – *Modelamento de Hardware Visando À Estimação do Tempo de Execução de Programas*. Dissertação de Mestrado apresentada ao Programa de Pós-Graduação em Engenharia Elétrica e Informática Industrial. CEFET-PR, Curitiba, 2001.

[Luguesi 06] Luguesi, J. L. – *Ambiente de Apoio ao Ensino e Aprendizado do Escalonamento em Sistemas em Tempo Real*. Dissertação de Mestrado apresentada ao Programa de Pós-Graduação em Engenharia Elétrica e Informática Industrial. UTFPR, Curitiba, Novembro, 2006.

[Macraigor 07] Macraigor Systems LLC – *Wiggler Data Sheet*. http://www.macraigor.com/downloads/wiggler_ds.pdf, Agosto, 2007.

[Mahrenholz 01] Mahrenholz, D. – *Minimal Invasive Monitoring*. Em Proceedings of the Fourth International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC'01). Pág. 0251, Maio/2001. IEEE Computer Society.

[Mattolini 01] Mattolini, R.; Nesi, P. – *An Interval Logic for Real-Time System Specification*. Em IEEE Transactions on Software Engineering. Pág. 208-227, v. 27, n. 3, Março/2001. IEEE Computer Society.

[Myers 89] Myers, G. – *The Art of Software Testing*. Editora Wiley, 1979.

[Navarro 06] Navarro, C. – *PET-IV: Ambiente Didático Interativo de Visualização de Sistemas Concorrentes Operando em Tempo Real*. Dissertação de Mestrado apresentada ao Programa de Pós-Graduação em Engenharia Elétrica e Informática Industrial. UTFPR, Curitiba, 2006.

[Nelson 90] Nelson, V. P.; – *Fault-Tolerant Computing: Fundamental Concepts*. Em IEEE Computer. Pág. 19-25, v. 23, n. 7, Julho/90. IEEE Computer Society.

[Nexus 03] *The Nexus 5001 Forum Standard for a Global Embedded Processor Debug Interface, Version 2.0*. 166 páginas, 2003. IEEE-ISTO.

[Opersys 07] Opersys Inc., *Linux Trace Toolkit Home Page*. http://www.opersys.com/LTT, Consultada em Agosto de 2007.

[Pinho 03] Pinho, A. C.; Cadamuro Jr., J.; Mori, F. Y.; Imay, M. C. – *Confiabilidade de Hardware em Projetos Complexos de Sistemas Computacionais Embarcados*. Em Anais do III Congresso Brasileiro de Computação (CBComp 2003), Pág. 729-739, Itajaí, SC, Agosto/2003.

[Power 06] *Power ISA Version 2.03 Specification*. International Business Machines (IBM), 828 páginas, Setembro/2006.

[Pressman 95] Pressman, R. S. – *Engenharia de Software*. Editora Makron Books do Brasil – São Paulo, 1995.

[Puschner 89] Puschner, P.; Koza, C. – *Calculating the Maximum Execution Time of Real Time Programs*. Real-Time Systems, v. 1, n. 2, pág. 159-176.

[Renaux 93] Renaux, D. P. B. – *RTX-Parlog: A Concurrent Logic Programming Language for Real-Time Systems*. Tese de Doutorado apresentada à Universidade de Waterloo. Canadá, 1993.

[Renaux 96] Renaux, D. P. B. – *PET – A Small Real-Time Support System for Microcontroller without Virtual Memory*. Relatório Técnico, CPGEI. CEFET-PR, Curitiba, Maio/1996.

[Renaux 99] Renaux, D. P. B.; Braga, A.; Kawamura, A. – *PERF: Um Ambiente para Avaliação Temporal de Sistemas em Tempo Real*. II Workshop de Sistemas em Tempo Real, Salvador. Pág. 76-87, 1999.

[Renaux 01] Renaux, D. – *Processo de Desenvolvimento de Sistemas em Tempo Real*. Relatório Interno, LIT/CEFET-PR, Out/2001.

[Renaux 02] Renaux, D. P. B.; Góes, J.; Linhares, R. – *WCET Estimation From Object Code Implemented in the PERF*. Em Proceeding of the Second International Workshop on Worst Case Execution Time Analysis, evento satélite ao 14th Euromicro Conference on Real-Time Systms. Viena, Áustria. Pág. 28-35, 2002.

[Schneier 94] Schneier, B.; – *Description of a New Variable-Length Key, 64-bit Block Cipher (Blowfish)*. Em Cambridge Security Workshop Proceeding. Pág. 191-204, 1994. Springer-Verlag.

[Schroeder 95] Schroeder, B. A.; – *On-Line Monitoring: A Tutorial*. Em IEEE Computer. Pág. 72-78, v. 28, n. 6, Junho/95. IEEE Computer Society.

[Sommerville 07] Sommerville, I. – *Engenharia de Software*. Editora Pearson, 8ª Edição, Brasil, 2007.

[Sowmya 98] Sowmia, A.; Ramesh, S. – *Extending Statecharts with Temporal Logic*. Em IEEE Transactions on Software Engineering. Pág. 216-231, v. 24, n. 3, Março/1998. IEEE Computer Society.

[Stunkel 91] Stunkel, C. B.; Janssens, B.; Fuchs, W. K. – *Address Tracing for Parallel Machines*. Em IEEE Computer. Pág. 31-38, v. 24, n. 1, Janeiro/91. IEEE Computer Society.

[Tektronix 05] *Tektronix Logic Analyzers – Family Selection Guide*. Catálogo de Produtos Tektronix. Documento 52W-13957-9. Tektronix, Inc., E.U.A., Junho/2005.

[Thane 00] Thane, H.; Hansson, H. – *Using Deterministic Replay for Debugging Distributed Real-Time Systems*. Nos anais de 12th Euromicro Conference on Real-Time Systems (Euromicro-RTS 2000). Pág. 265, Junho/2000.

[Thane 03] Thane, H.; Sundmark, D.; Huselius, J.; Petterson, A. – *Replay Debugging of Real-Time Systems Using Time Machines*. Em Proceedings of the International Parallel and Distributed Processing Symposium (IPDPS'03). Pág. 288b, Abril/2003. IEEE Computer Society.

[Tsai 90] Tsai, J. J. P.; Fang, K.; Chen, H; Bi, Y. – *A Noninterference Monitoring and Replay Mechanism for Real-Time Software Testing and Debugging*. Em IEEE Transactions on Software Engineering. Vol. 16, nº 8, Págs. 897-916, Agosto/1990.

[Tsai 96] Tsai, J. J. P.; Bi, Y.; Yang, S. J. – *Debugging for Timing-Constraint Violations*. Em IEEE Software. Págs. 89-99, Março/1996. IEEE Computer Society.

[Turley 03] Turley, J. – *The Two Percent Solution*. Embedded Systems Programming – v. 16, n. 1 – Janeiro/2003.

[Turley 04] Turley, J. – *Nexus Standard Brings Order to Microprocessor Debugging*. Documento aberto do Forum Nexus 5001, http://www.nexus5001.org, 2004. The Nexus 5001 Forum.

[Wilner 95] Wilner, D. – *WindView: A Tool for Understanding Real-Time Embedded Software Through System Visualization*. Em II ACM SIGPLAN Workshop on Languages, Compilers and Tools for Real-Time Systems. Orlando, Flórida, E.U.A., Junho/1995. Association for Computing Machinery.

[Wind 95] *VxWorks 5.3 Programmer's Guide*. Manual do Tornado. Documento DOC-11045-ZD-01, 1ª Edição. Wind River Systems, Inc., E.U.A., Dezembro/1995.

[Wind 99] *WindView 2.0.1 User's Guide*. Manual do Tornado II. Documento DOC-12393-ZD-02, 1ª Edição. Wind River Systems, Inc., E.U.A., Março/1999.

ANEXO I: O NÚCLEO DE TEMPO REAL PET#

O PET# é a versão mais recente da família de núcleos de tempo real PET. A primeira versão do PET foi liberada em 1995 para uso acadêmico, e no ano seguinte para uso industrial [Renaux 96]. Ambas foram escritas para microcontroladores Intel 80186. Algum tempo depois o núcleo foi portado também para processadores Motorola PowerPC MPC8xx [Cadamuro 01], e para microcontroladores ARM (*Advanced RISC Machine*), na variante ARM7TDMI.

A família de núcleos de tempo real PET, desde sua primeira versão, foi construída levando em consideração as características requeridas e esperadas dos sistemas embarcados operando em tempo real, dentre as quais podemos destacar: grandes e complexos; confiáveis e seguros; controle concorrente de diferentes componentes do sistema; tempos de resposta rigidamente definidos; altamente acoplado com o ambiente externo; ambiente de execução eficiente [Burns 97].

A família PET é baseada em princípios de *microkernel*, ou seja, o núcleo é composto por poucos serviços que são implementados de forma eficiente e testados intensivamente. Deste modo, o núcleo tende a ter menos erros, tempos de resposta conhecidos e boa estabilidade. No sistema baseados na arquitetura de *microkernel* os serviços mais complexos são implementados na forma de componentes de *software* que residem no espaço do usuário, o que provê escalabilidade para aplicações com diferentes tamanhos e necessidades.

A PET cria para o usuário um ambiente multiprogramado onde todas as tarefas do usuário compartilham o mesmo espaço de endereçamento. Esta abordagem aumenta a eficiência do sistema, reduzindo o tempo gasto com cópias de informações entre diferentes tarefas e permitindo comutações de contexto computacionalmente mais baratas. Além do mais, não existem prejuízos para a aplicação na maioria dos casos, pois os sistemas embarcados são, tipicamente, cooperativos.

A comunicação e a sincronização entre as tarefas do PET são feitas com mensagens. Segundo Burns [Burns 97] existem três modelos de trocas de mensagens entre tarefas, que podem ser descritos em função do comportamento do remetente. O primeiro modelo é o envio assíncrono, onde o remetente continua a executar após o envio, independente de a mensagem ter sido entregue ao destinatário ou não. O segundo modelo é o envio síncrono, onde o remetente é bloqueado no momento do envio, permanecendo neste estado até que o receptor receba a mensagem. O terceiro modelo é conhecido como chamada remota e requer

que o destinatário, além de receber a mensagem, também responda ao remetente para que este seja desbloqueado. Todas as versões do PET implementam tanto o envio assíncrono como a chamada remota. Entretanto, nas versões anteriores, mensagens assíncronas tinham precedência sobre as chamadas remotas, ou seja, ainda que houvessem diversas mensagens enviadas por chamada remota, a chegada de uma mensagem assíncrona era atendida por primeiro. No PET# todas as mensagems enviadas para uma tarefa têm a mesma prioridade e são enfileiradas por ordem de chegada.

Outra diferença entre o PET# e seus antecessores na família PET com relação a troca de mensagens é o fato destas, que antes eram objetos definidos pelo usuário, agora serem objetos estruturados, de tamanho fixo e definidos pelo sistema. Estes objetos são compostos por um comando e cinco argumentos. A nova abordagem apresenta a vantagem de tornar determinístico o tempo efetivamente gasto com o envio ou recebimento de uma mensagem. No entanto, a responsabilidade por transferências de dados maiores do que os argumentos da mensagem é passada para o usuário, o que não apresenta um problema significativo uma vez que todas as tarefas do usuário executam no mesmo espaço de endereçamento.

Para atender a requisitos de tempos de execução mais estritos, o PET# optou por bloquear serviços que possuam tempo de execução não-determinístico durante a operação normal do sistema. Nesta categoria estão incluídas a criação e destruição de tarefas, e a alocação dinâmica de memória. Usando esta abordagem todas as tarefas precisam ser criadas durante a inicialização do sistema, e sua implementação esperada é um laço infinito. Se por qualquer motivo uma tarefa sair da sua função principal, o PET# é capaz de detectar tal situação e liberar tanto o descritor da tarefa quanto a memória utilizada por ela de forma adequada, mas na maioria dos casos isto é considerado um término anormal. A alocação dinâmica de memória também é um serviço que fica disponível durante a inicialização do sistema, mas seu uso após o início do escalonamento não é recomendado porque rotinas para desfragmentar ou para descobrir e remover trechos de memória alocados e não desalocados possuem tempos de execução muito variáveis.

O PET# foi escrito em linguagem C, ao contrario das versões anteriores que haviam sido escritas em linguagem C++. O ambiente de desenvolvimento utilizado para gerar o PET# e a aplicação é o MULTI 2000, da Green Hills Software [Green 07]. Três motivos principais levaram ao uso do MULTI 2000, ao invés de um ambiente de *software* livre. Primeiro, o compilador de última geração do MULTI. Segundo, a disponibilidade de depuração com ferramentas JTAG/OCD de baixo custo. Terceiro, a biblioteca ANSI C bem

documentada, adaptável para diferentes sistemas operacionais e com suporte para utilização em sistemas concorrentes.

O escalonador do PET# é baseado em prioridades e preemptivo. Tarefas que estejam no mesmo nível de prioridade são escalonadas pelo algoritmo FIFO (*First In, First Out*), assumindo que o sistema é cooperativo. A inclusão da preempção no escalonador pode ser considerada a diferença mais relevante entre o PET# e seus antecessores.

O PET# foi concebido para suportar rastreabilidade em conformidade com os requisitos do Dyretiva. Para tanto, uma macro de instrumentação foi acrescentada no código nos pontos onde informações relevantes sobre o estado das tarefas ou do sistema operacional são processadas. Dependendo dos recursos disponíveis no *hardware*, a macro pode ser vazia, implementar monitoração baseada em *software*, ou escrever na porta de medição. A monitoração baseada em *software* pode ser utilizada em testes de tarefas individuais ou em testes de comportamento (Seção 2.2.2) de sistemas legados ou que utilizam *hardware* padrão O novos projetos podem includir no *hardware* o suporte para a porta de medição definida pelo Dyretiva, permitindo o uso das ferramentas do SoftScope para em verificações e testes que sejam necessários.

SERVIÇOS DO PET#

Os serviços básicos oferecidos pelo PET# podem ser divididos em quatro categorias: troca de mensagens, manipulação de interrupções de *hardware*, tempo e gerenciamento de tarefas. Um breve resumo dos serviços básicos do PET# é apresentado na Tabela 15. A coluna esquerda da Tabela 15 mostra a assinatura de cada serviço no estilo da linguagem C, mas sem especificar a maioria dos tipos envolvidos. Já na coluna direita da Tabela 15 existe uma breve descrição de cada serviço.

Os serviços de gerenciamento de tarefas são utilizados para criar tarefas durante a inicialização do sistema, e para ler o identificador de uma tarefa durante a operação. A execução destes serviços não é reportada ao sistema de monitoração, pois não mudam o estado das tarefas ou fornecem informações relevantes para a análise do arquivo de rastreamento.

Tabela 15: Lista de serviços básicos oferecidos pelo PET#.

Serviço	Descrição
Serviços de Gerenciamento de Tarefas	
CreateThread (nome, prio, tam_pilha, opcoes, &tid, funcao, par1, par2)	Cria uma nova tarefa no sistema
tid = GetThreadId(nome)	Lê o identificador de uma tarefa
tid = GetMyThreadId( )	Lê o identificador da própria tarefa
GetThreadName(tid, buf, tam_buf)	Lê o nome de uma tarefa
Serviços de Tempo	
SetTime(segundos)	Atualiza data e hora do sistema
GetTime(time)	Lê data e hora do sistema
Sleep(miliseg)	Bloqueia uma tarefa por um número de milisegundos
SetTimer(miliseg, modo, comando)	Programa um temporizador para uma tarefa
CancelTimer(comando)	Cancela um temporizador previamente programado
Serviços de Troca de Mensagens	
PutMessage(dest, msg)	Envia uma mensagem assíncrona
SendMessage(dest, msg)	Inicia uma chamada remota
ReceiveMessage(msg, &remet)	Recebe uma mensagem, ou bloqueia
ReplyMessage(remet, msg)	Responde a uma mensagem numa chamada remota
Serviços de Manipulação de Interrupções de *Hardware*	
RegisterISR (irq_id, manip, param, config)	Registra um manipulador do usuário para atender a uma interrupção de *hardware*
UnregisterISR(irq_id)	Libera uma interrupção de *hardware*
MaskIRQ(irq_id)	Mascara uma interrupção no controlador de interrupções
UnmaskIRQ(irq_id)	Desmascara uma interrupção
estado = IsMasked(irq_id)	Verifica se uma interrupção está mascarada
estado = IsPending(irq_id)	Verifica se uma interrupção está aguardando atendimento
flags = LockIRQS( )	Desabilita as interrupções no processador
UnlockIRQS(flags)	Recupera o estado das interrupções no processador
PutMessageISR(dest, msg)	Envia uma mensagem de dentro de uma rotina de atendimento para uma tarefa

Os serviços de tempo servem para gerenciar data e hora do sistema, atrasar a execução de uma tarefa ou utilizar temporizadores de *software* oferecidos pelo sistema operacional. Todos estes serviços são implementados por uma tarefa (TimeManager), e não pelo núcleo em si. Quando um dos serviços de tempo é chamado pelo usuário, uma biblioteca do PET# converte a chamada da função em uma chamada remota ao TimeManager. Cada chamada a um serviço de tempo gera um único evento para o sistema de monitoração para permitir que a precisão das temporizações possa ser avaliada posteriormente.

Os serviços de troca de mensagens formam a parte central do PET#. Eles são implementados como chamadas de sistema (*system calls*), e têm a capacidade de provocar reescalonamento. Existem quatro razões para que um novo escalonamento seja invocado durante um envio ou recebimento de mensagem. Primeiro quando num envio assíncrono o destinatário tem prioridade mais alta que o remetente. Segundo quando a mensagem que inicia uma chamada remota é enviada. Terceiro quando se solicita o recebimento de uma mensagem, mas não existe uma disponível. Quarto e último, quando a resposta a uma chamada remota é evniada, e o cliente tem prioridade mais alta que o servidor. Cada serviço de mensagem é instrumentado em exatos três pontos: no início da chamada de sistema, na indicação do serviço e no final da chamada de sistema.

Para o PET# as tarefas do usuário podem estar em um dos quatro estados possíveis: pronta, executando, bloqueada em transmissão ou bloqueada em recepção. As mudanças de estado acontecem devido às trocas de mensagens. Um diagrama de estados, em Statecharts [Harel 87], das transições de estado nas tarefas do PET# é mostrado na Figura 52.

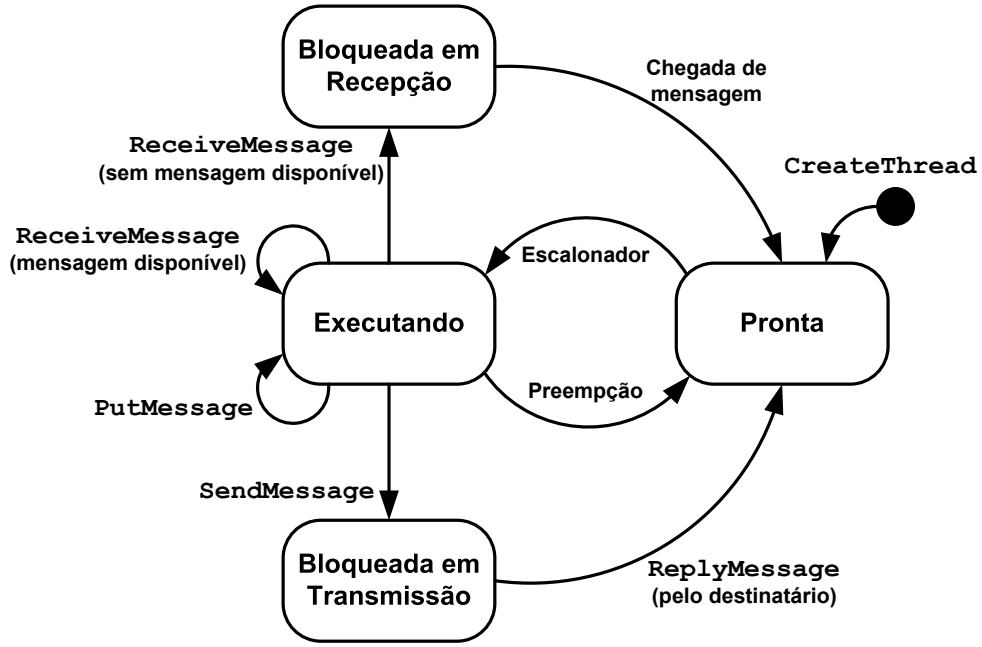


Figura 52: Diagrama de estados de uma tarefa no PET#.

Os serviços de manipulação de interrupções de *hardware* do PET# oferecem ao usuário uma visão consistente dos recursos de interrupção do processador, independentemente da arquitetura utilizada. Existem serviços para registrar manipuladores de interrupção dos dispositivos controlados pelo usuário, manipular o(s) controlador(es) de interrupção, inibir as interrupções no processador a enviar mensagens de dentro de rotinas de atendimento para tarefas. Este último serviço pode provocar a preempção da tarefa que está executando, tornando possível a preempção de qualquer tarefa em qualquer ponto da execução. Os serviços de manipulação de interrupções, visíveis ou não, são instrumentados de forma que o sistema de rastreamento do Dyretiva possa ter as informações necessárias para identificar e calcular as latências causadas pelas interrupções de *hardware*. Estas classe de serviços não estava disponível nas versões anteriores do PET, mas foi acrescentada no PET# para atender as necessidades de portabilidade de programas escritos para PET entre diferentes plataformas.

ANEXO II: O PROTOCOLO DE COMUNICAÇÃO DO MIMO

O protocolo de comunicação utilizado pelo MIMO é do tipo mestre-escravo e ponto-a-ponto, onde o programa de controle do monitor é o mestre e o MIMO é o escravo. Toda comunicação é iniciada pelo mestre, seguida de uma resposta do escravo.

O protocolo é baseado em mensagens enviadas em pacotes de bytes cujo formato é mostrado na Figura 53.

Figura 53: Formato da mensagem do protocolo do MIMO.

Com exceção do campo DADO, que pode possuir até 255 bytes de comprimento, todos os outros campos da mensagem possuem apenas um byte. Se qualquer informação dentro do campo DADO for maior do que um byte, sua interpretação deve ser feita no formato *big endian*, ou seja, com os bytes mais significativos da palavra sendo enviados primeiro. Uma descrição de cada campo da mensagem no protocolo MIMO é apresentada na Tabela 16.

Tabela 16: Descrição dos campos contidos na mensagem do protocolo MIMO.

Campo	Descrição
SOM	Marcador de início da mensagem. Possui valor fixo 0x55.
CMD	Comando sendo solicitado ou respondido na mensagem.
SEQ	Número de seqüência da mensagem.
LEN	Número de bytes contidos no campo DADO.
DADO	Conteúdo da mensagem.
CKS	*Checksum* da mensagem, computado desde o campo CMD até o último byte do campo DADO.
EOM	Marcador de final da mensagem. Possui valor fixo 0x00.

Pelo protocolo do MIMO, uma mensagem sempre se inicia com o valor SOM (0x55), que não deve aparecer em nenhum outro campo da mensagem sendo transmitida. No entanto, como outros campos da mensagem podem ter valor 0x55, existe um caractere de escape definido, com valor 0xDF, para contornar esta questão. Assim, quando o valor 0x55 é encontrado em um campo diferente do SOM na mensagem sendo transmitida, ele será substituído pelo caractere de escape (0xDF), seguido do valor de escape do SOM, que é

0xD5. Desta forma, quando o receptor identifica os valores 0xDF e 0xDE em uma mensagem recebida, ele faz a operação contrária, recuperando o valor 0x55 como parte da mensagem original e descartando os bytes 0xDF e 0xDE. Para evitar dualidade na interpretação do escape, quando um valor de escape (0xDF) é encontrado na mensagem sendo transmitida, um outro caractere de escape é acrescentado para que o receptor saiba a real intenção do transmissor. Este caractere extra é removido durante a recepção. Um exemplo de transmissão de uma mensagem que contem tanto o caractere 0x55 como o caractere 0xDF é ilustrado na Figura 54.

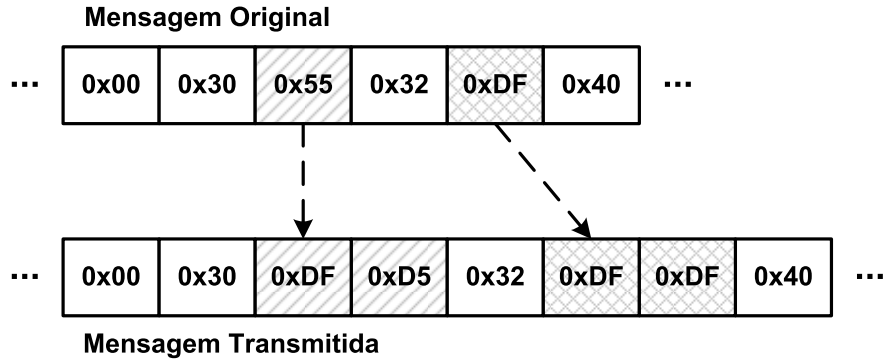


Figura 54: Exemplo de transmissão de mensagem contendo caracteres SOM e escape.

O próximo campo da mensagem no protocolo do MIMO após o SOM é o CMD, cujo byte é subdividido em três campos, conforme mostrado na Figura 55.

Figura 55: Conteúdo do campo CMD na mensagem do protocolo MIMO.

Na Figura 55 os seis bits que compõe o campo Comando contém um valor que representa um dos seis comandos existentes no MIMO: ecoar, ler configuração, iniciar amostragem, finalizar amostragem, ler eventos e reiniciar. Este campo precisa ser preenchido tanto pelo mestre, na solicitação da operação, como pelo escravo, na resposta ao comando. Os valores possíveis para este campo do CMD são apresentados na Tabela 17.

Tabela 17: Valores para o campo Comando do byte CMD da mensagem.

Comando	Valor
Ecoar	0
Ler configuração	1
Iniciar amostragem	2
Finalizar amostragem	3
Ler eventos	4
Reinicializar	5

Quando o mestre faz uma solicitação ao escravo, os bits ACK e NACK (bits 7 e 6, respectivamente, da Figura 55) do byte CMD devem ser preenchidos com zero. Entretanto, quando o escravo responde a mensagem ele deve preencher um destes dois campos com o valor um, de modo a informar ao mestre se o comando foi recebido e executado com sucesso (ACK), ou se a mensagem chegou corrompida e precisa ser retransmitida (NACK).

O campo seguinte no protocolo do MIMO é o número de seqüência (SEQ) da mensagem. Este número identifica uma determinada instância de uma solicitação, sendo preenchido pelo transmissor e utilizada pelo receptor para saber se um determinado comando já foi recebido e executado anteriormente ou não. Utilizando o número de seqüência o escravo é capaz de descartar mensagens enviadas em duplicidade pelo mestre, bem como reenviar respostas que tenham sido perdidas na linha de transmissão. Quando o mestre gera o número de seqüência para um novo comando, a única restrição que existe é que ele não seja igual ao valor utilizado na última solicitação. Do lado do escravo, se um comando é recebido com o mesmo número de seqüência do último comando recebido, então apenas o ACK é enviado novamente ao mestre.

O próximo campo da mensagem no protocolo do MIMO é o LEN, que informa ao receptor da mensagem quantos bytes existem para ser recebidos no campo DADO, que vem logo a seguir. Após a recepção de todos os dados, o transmissor envia o *checksum* da mensagem, que é computado desde o campo CMD até o último byte de DADO. O valor recebido pelo receptor deve ser verificado contra o valor calculado ao longo do recebimento e, caso estes valores sejam diferentes, ações corretivas (enviar NACK no caso do escravo, ou retransmitir mensagem no caso do mestre) precisam ser tomadas no sentido de recuperar a comunicação. Para marcar claramente o final da mensagem, um byte de EOM (0x00) é enviado logo após o CKS. Com isto o receptor pode sincronizar sua máquina de estados de recepção e preparar-se para receber o próximo SOM.

Para prevenir-se contra perdas ou corrupção das mensagens, o mestre sempre envia uma solicitação ao escravo e aguarda até um segundo pela resposta. Se a resposta não vier neste tempo, três novas tentativas de envio da mesma solicitação são feitas seguindo o mesmo princípio. Em caso de ainda assim não haver resposta, o envio é cancelado e um erro é retornado ao nível superior do protocolo. Se, por outro lado, o mestre receber uma resposta com erro de *checksum*, esta resposta é descartada, pois o reenvio posterior da solicitação obrigará o escravo a responder novamente.