UNIVERSIDADE FEDERAL DE PERNAMBUCO
CENTRO DE INFORMÁTICA
PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

VANESSA LARIZE ALVES DE CARVALHO

VALIDAÇÃO DE UMA ESPECIFICAÇÃO TDEVC PARA O DESENVOLVIMENTO DE DEVICE DRIVERS ROBUSTOS

Recife
2016

VANESSA LARIZE ALVES DE CARVALHO

VALIDAÇÃO DE UMA ESPECIFICAÇÃO TDEVC PARA O DESENVOLVIMENTO DE DEVICE DRIVERS ROBUSTOS

Trabalho apresentado ao Programa de Pós-graduação em Ciência da Computação do Centro de Informática da Universidade Federal de Pernambuco como requisito parcial para obtenção do grau de Mestre em Ciência da Computação.

Orientador: *Prof^a. Dr^a. Edna Natividade da Silva Barros*

Recife
2016

Catalogação na fonte
Bibliotecária Joana D'Arc Leão Salvador CRB 4-572

C331v Carvalho, Vanessa Larize Alves de.
Validação de uma especificação TDEVC para o desenvolvimento de device drives robustos / Vanessa Larize Alves de Carvalho. – 2016.
84 f.: fig., tab. quadros.

Orientadora: Edna Natividade da Silva Barros.
Dissertação (Mestrado) – Universidade Federal de Pernambuco. CIN. Ciência da Computação, Recife, 2016.
Inclui referências.

1. Engenharia da computação. 2. Validação de sistemas. 3. Sistema embarcados. 4. Provadores automáticos de teoremas. I. Barros, Edna Natividade da Silva (Orientadora). II. Titulo.

621.39 CDD (22. ed.) UFPE-MEI 2017-15

VANESSA LARIZE ALVES DE CARVALHO

Validação de uma Especificação TDevC para o Desenvolvimento de Device Drivers Robustos

Dissertação apresentada ao programa de Pós-Graduação em Ciência da Computação do Centro de Informática da Universidade Federal dePernambuco, como requisito parcial para obtenção do título de Mestre em Ciência da Computação.

Aprovada em: 15/09/2016

BANCA EXAMINADORA

Prof^a. Dr^a. Edna Natividade da Silva Barros
Centro de Informática/UFPE
(**Orientador**)

Prof. Dr. André Aziz Camilo de Araujo
Departamento de Estatística e Informática/UFRPE

Prof. Dr. Giovanny Fernando Lucero Palma
Departamento de Ciência da Computação e Estatística/UFS

Recife
2016

Dedico essa dissertação aos meus avôs, Laninho e Virgínia, por toda dedicação e incentivo. Vocês sempre estarão em meus pensamentos.

Agradecimentos

Finalizar mais um desafio da minha vida não foi fácil, mas sem a ajuda de algumas pessoas seria infinitamente mais difícil. Primeiramente gostaria de agradecer a Deus e à minha base, minha mãe Marlene e minhas irmãs Lígia e Geórgia, vocês são a minha inspiração diária. A Gabriel Di Atlanta por estar sempre ao meu lado, me fazendo acreditar que posso mais do que imagino. À professora Edna Barros pela orientação e ao professor Alexandre Mota pela disponibilidade e prontidão em sanar as minhas dúvidas.

Gostaria de agradecer também a Rafael Macieira por todas as explanações, pela disponibilidade e paciência durante esses dois anos de pesquisa. À Camila Ascendina por todo o suporte, palavras positivas e por me fazer sempre seguir em frente, você é uma pessoa que emana compaixão.

E para finalizar, gostaria de agradecer a Jéssyca, por ter me acolhido em sua residência em um momento delicado, e a todos os meus amigos de Petrolina, do Recife e do Centro de Informática (UFPE), em nome de Nair, Laisy, Marianne, Crystal e Eduardo, por todos esses anos de amizade e por trazerem mais leveza a essa dura jornada.

Não acreditem no que eu digo, testem por si próprios.

—BUDA SAKIAMUNI

Resumo

O uso de sistemas eletrônicos embarcados está cada vez mais presente no dia a dia da sociedade. Telefones celulares,sistemas de posicionamento global (GPS) e televisores digitais com telas de LCD são exemplos de equipamentos que estão incorporando funcionalidades para atender às demandas dos usuários, e, consequentemente, aumentando a complexidade dos sistemas embarcados nesses dispositivos. De fato, a grande maioria das inovações em sistemas embarcados é atribuída aos avanços na microeletrônica e no projeto de software embarcado. Porém, devido a atual complexidade dos dispositivos, projetos de hardware não conseguem acompanhar o crescimento da capacidade física do hardware, havendo um gap de produtividade entre o desenvolvimento do hardware e o desenvolvimento do software necessário para a sua operação. Esses softwares, também conhecidos como Software dependente do Hardware (Hardware-dependent Software - HdS), estão no centro do desafio do projeto de sistemas. Dentre esses HdS, pode-se citar os device drivers ou drivers dos dispositivos. Os drivers são codificados com base na documentação disponível pelos fornecedores do hardware, porém, na maioria das vezes, esse documento não é de fácil leitura, podendo levar a erros de interpretação. Atrelado a isso, como essa documentação está escrita em uma linguagem natural, a descrição do dispositivo pode ser muitas vezes ambígua, incompleta, ou até mesmo inconsistente. Além desses problemas, o device driver tem acesso a vários recursos do sistema operacional, assim qualquer erro nesta camada de software pode ser fatal. Por isso, essa camada de software deve ser cuidadosamente desenvolvida e testada. Com o intuito de reduzir os erros nos devices drivers, em MACIEIRA; BARROS; ASCENDINA (2014) foi proposta uma técnica de formalização e validação em tempo de execução de propriedades temporais e protocolos de comunicação de alto nível entre os dispositivos e seus devices drivers utilizando a linguagem TDevC. Mas, na especificação do trabalho anterior, a máquina de estados hierárquica gerada ainda pode conter estados não-determinísticos e propriedades temporais contraditórias. Dessa forma, o presente trabalho propõe uma técnica para validação de uma especificação TDevC para o desenvolvimento de device drivers robustos. Para isso, este trabalho faz uso do provador de teoremas de alto desempenho Z3 e das propriedades dos autômatos de Büchi. Para validação da proposta, foi utilizada a especificação TDevC do dispositivo Ethernet DM9000A.Nos experimentos realizados, verificou-se que o projeto conseguiu detectar as inconsistências na especificação TDevC em 100% dos casos.

Palavras-chave: Software Dependente do Hardware - HdS. Validação de uma especificação TDevC. Provador automático de teoremas. Autômatos de Büchi.

Abstract

The use of electronic embedded system has increased substantially. Mobile phones, Global Positioning System (GPS) and Digital television with LCD screens are examples of equipments that are incorporating features to meet the demands of users, and thereby increasing the complexity of embedded systems in these devices. In fact, the vast majority of innovations in embedded systems is attributed to advances in microelectronics and embedded software design. However, due to the current complexity of devices, hardware design cannot keep up the hardware capacity growth, with a productivity gap between the development of the hardware and the development of the software required for its operation. These softwares, also known as Hardware-dependent Software (HdS) are at the center of the design challenge systems. Among these HdS are the devices drivers. Drivers are encoded based on the documentation available by the hardware vendors, however, most of the time, this document is not easy to read and can lead to misinterpretations. Coupled to this, as this documentation is written in a natural language, the device description can often be ambiguous, incomplete or even inconsistent. In addition to these problems, the device driver has access to various operating system resources, so any error in this software layer can be fatal. Therefore, this software layer must be carefully developed and tested. In order to reduce errors in the device drivers, it has been proposed a technique for formalization and runtime validation of temporal properties in high-level communication protocols between devices and drivers using the TDevC language. But the hierarchical state machine, generate in the previous work, may contain nondeterministic states and contradictory temporal properties. Thus, this approach proposes a technique to validate a TDevC specification for the development of robust device drivers. Therefore, this work makes use of high-performance theorem prover Z3 and Buchi automata properties. Some experiments using the Ethernet device DM9000A TDevC specification showed that this approach is effective in detect TDevC specification inconsistency.

Keywords: Hardware-dependent Software. Validation of a TDevC specification. High-performance theorem prover. Büchi automata.

Lista de Figuras

Lista de Quadros

Lista de Acrônimos

BA	Autômato de Büchi (Büchi Automaton)
DSL	Domain-Specific Language
CPU	Central Processing Unit
HFSM-D	Hierarchical Finite-State Machine with Datapath
LTL	Linear Temporal Logic
MDDC	Monitor of Driver/ Device Communication

Sumário

1

Introdução

O uso de sistemas eletrônicos embarcados está cada vez mais presente no dia a dia da sociedade. Telefones celulares, sistemas de posicionamento global (GPS) e televisores digitais com telas de LCD são exemplos de equipamentos que estão incorporando funcionalidades para atender às demandas dos usuários, e consequentemente aumentando a complexidade dos sistemas embarcados nesses dispositivos.

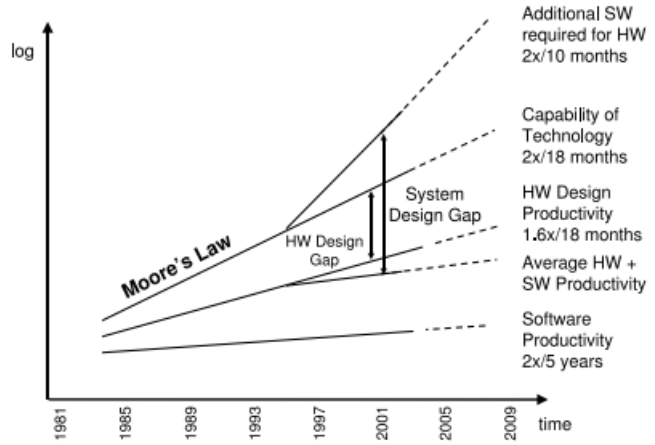
De fato, a grande maioria das inovações em sistemas embarcados é atribuída aos avanços na microeletrônica e no projeto de software embarcado. Em particular, de acordo com a lei de Moore, o número de transistores que podem ser integrados em um único chip dobra a cada 18 meses MOORE (1965). Porém, devido a atual complexidade dos dispositivos, projetos de hardware não conseguem acompanhar essa tendência, tendo apenas um crescimento de 1,6 vezes a cada 18 meses, crescimento esse atribuído ao uso de núcleos de propriedade intelectual (IP cores)ECKER; MüLLER; DöMER (2009). Uma vez que a capacidade do hardware está crescendo mais rapidamente do que a produtividade, pode-se verificar o gap no projeto de hardware, visto na figura 1.1. Para piorar esse cenário, a produtividade no projeto de software cresce aproximadamente 2 vezes a cada 24 meses, no entanto, para satisfazer as necessidades reais em termos de complexidade de software embarcado, seria necessário um crescimento de aproximadamente 2 vezes a cada 10 meses.

Assim, em adição ao gap no projeto de hardware, também estamos diante de um gap no projeto de software. Considerando os dois problemas juntos, como visto na figura 1.1, os dois gaps combinados resultam em um grande gap no projeto de sistemas.

É necessário enfatizar que o real desafio do projeto de sistemas embarcados não é somente a adição de dois gaps de projeto, mas também a estreita e grande dependência entre os domínios de software e hardware. Em outras palavras, a interface necessária entre o software e o hardware adiciona outra camada de complexidade. Desse modo, Software dependente de Hardware (*Hardware-dependent Software - HdS*) está no centro desse desafio de projeto de sistemas ECKER; MüLLER; DöMER (2009).

Software dependente de Hardware é um software em um sistema embarcado que interage diretamente com a plataforma de hardware subjacente, sendo crítico para o funcionamento do

Figura 1.1: Gap entre projetos de hardware, software e sistema



Fonte: ECKER; MüLLER; DöMER (2009)

sistema. Além disso,ECKER; MüLLER; DöMER (2009):

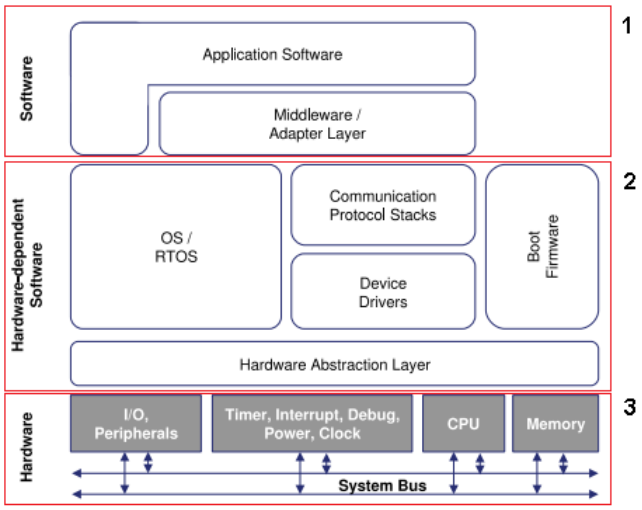
- HdS é construído especificamente para um bloco de hardware em particular, ou seja, o HdS não tem sentido sem o harware;
- HdS e hardware juntos implementam a funcionalidade do sistema, ou seja, o hardware não tem sentido sem o HdS;
- HdS fornece uma interface para facilitar o acesso os recursos do hardware.

Atualmente, em grande parte de projetos interdisciplinares, a importância do HdS ainda não é entendida. Frequentemente, a necessidade de HdS é subestimada na fase de planejamento do projeto e algumas vezes completamente ignorada ECKER; MüLLER; DöMER (2009). Em contraste, HdS se torna um fator de crucial importância quando o projeto avança e, quando problemas são descobertos tardiamente, geralmente a reparação torna-se mais cara.

O fato do comportamento e da implementação do HdS estarem diretamente relacionados ao comportamento do hardware do sistema tornam a sua codificação extremamente difícil e susceptível à falhas. Além disso, a inserção de erros nesse componente de software embarcado pode ser catastrófica, ou seja, possui uma alta criticidade.

A figura 1.2 ilustra os diversos HdS em uma arquitetura de software em camadas. No topo da figura, parte 1, encontram-se as aplicações que são suportadas pelo HdS. No meio, parte 2, têm os vários tipos de HdS, que tipicamente são executados no modo kernel do sistema operacional. Essa camada pode incluir os seguintes componentes de software: Sistema Operacional ou Sistema Operacional de Tempo Real (RTOS), código de inicialização do sistema, pilhas de protocolos de comunicação e device drivers. E na parte 3 da figura, tem-se os diversos dispositivos ou periféricos.

Figura 1.2: HdS em uma arquitetura de software em camadas



Fonte:ECKER; MüLLER; DöMER (2009)

Mais especificamente, a arquitetura típica de um sistema embarcado é composta pelos seguintes componentes principais ECKER; MüLLER; DöMER (2009):

- Software da aplicação: composto por uma ou múltiplas aplicações que implementam as funcionalidades do sistema. Esses softwares podem ser compostos de múltiplos processos e/ou *threads*. Porém, na maioria dos sistemas embarcados, os softwares da aplicação servem a uma única aplicação com um propósito específico;
- Sistema operacional: é um componente de software que gerencia as tarefas do software da aplicação para o compartilhamento de recursos de hardware e software disponíveis. Se um sistema operacional suporta o compartilhamento de recursos sob restrições de tempo real, ele é chamado de sistema operacional de tempo real (RTOS);
- Pilha do protocolo de comunicação: os protocolos de comunicação são geralmente implementados por módulos de software em camadas em cima dos drivers dos dispositivos;
- Drivers de dispositivos: um driver de dispositivo fornece uma abstração em software para acessar os recursos de um hardware. A maioria dos drivers fornecem funcionalidades padrão para inicializar/reinicializar um dispositivo, ler e escrever fluxos de dados e realizar o controle de entrada e saída;
- Inicialização do *firmware*: o firmware de inicialização gerencia o processo de inicialização de um computador e tipicamente residem em uma memória só de leitura (ROM). Um exemplo de firmware de inicialização é o sistema básico de entrada e saída (BIOS);
- Camada de abstração do hardware: essa camada de software fornece uma interface abstrata para acessar os recursos do hardware.

Dentre esses componentes, os devices drivers merecem um especial destaque. Os device drivers, ou drivers de dispositivos, são programas que controlam um dispositivo em particular, implementando funcionalidades relativas ao controle, à comunicação e ao interfaceamento do dispositivo com o sistema operacional (ou diretamente com a aplicação em sistemas menores), traduzindo instruções de entrada e saída do sistema operacional em uma linguagem que um dispositivo possa entender, e vice versa. Ou seja, é uma camada de software que fica entre as aplicações e o dispositivo, fornecendo uma abstração para a sua manipulação CORBET; RUBINI; KROAH-HARTMAN (2005).

A camada de abstração do hardware (HAL) provê aos drivers do dispositivo (device drivers), ou simplesmente drivers, uma abstração dos dispositivos. É através dessa camada que os drivers fornecem, através de uma interface mais amigável, acesso aos recursos do hardware, traduzindo protocolos de comunicação entre os dispositivos e as funções de software localizadas nas mais altas camadas da pilha de software.

O projeto de um dispositivo está sujeito a diversas restrições, que muitas vezes são contraditórias, como requisitos de desempenho e compatibilidade com versões anteriores. Como resultado, a programação do interfaceamento com o dispositivo torna-se complexa e propensa a adição de erros. Além disso, os devices drivers são codificados com base na documentação disponível pelos fornecedores do hardware, porém, na maioria das vezes, esse documento não é de fácil leitura, podendo levar a erros de interpretação. Ademais, como essa documentação está escrita em uma linguagem natural, a descrição do dispositivo pode ser muitas vezes ambígua, incompleta, ou mesmo inconsistente REVEILLERE et al. (2000).

Como os drivers de dispositivos executam no modo *kernel*, ou modo supervisor, esses softwares têm permissão para acessar qualquer recurso em qualquer endereço. Assim, qualquer erro de programação pode comprometer todo o sistema. Além disso, tem-se que 70% do código do *kernel* (núcleo) do sistema operacional *Linux*, por exemplo, é composto por *device drivers*, os quais possuem uma taxa de erro de aproximadamente 3 a 7 vezes maior do que qualquer outro código do *kernel* CHOU et al. (2001). Dessa forma, para evitar que essas falhas ocorram, essa camada de software deve ser cuidadosamente desenvolvida e testada.

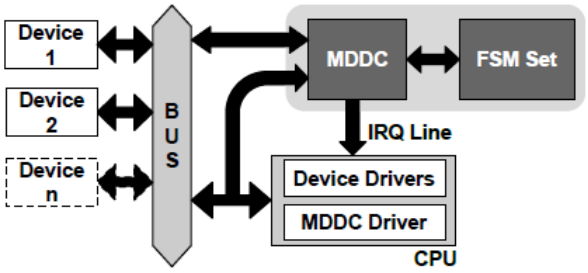
No domínio dos sistemas embarcados, como são projetados para plataformas de hardware específicas, cada sistema requisita novos *drivers* e/ou o porte de alguns já desenvolvidos, o que pode inviabilizar o reúso de *drivers* já implementados REVEILLERE et al. (2000). Em virtude disso, em muitos casos, os *drivers* são desenvolvidos a partir da cópia e edição de *templates* de códigos de *drivers* já existentes, na maioria das vezes sem o entendimento mais aprofundado, o que acarreta em erros que seriam facilmente evitados se eles fossem codificados do início, sem o uso de um código base.SWIFT et al. (2002)

Baseado nos problemas apresentados, relacionados com o desenvolvimento e execução de sistemas embarcados, o trabalho apresentado em MACIEIRA; BARROS; ASCENDINA (2014) propõe uma técnica que auxilie o desenvolvimento de *device drivers* robustos e confiáveis. O mecanismo proposto naquele trabalho realiza o monitoramento de propriedades temporais

descritas em modelos de alto nível de abstração através de monitores sintetizados e integrados em plataformas virtuais.

A arquitetura do mecanismo proposto em MACIEIRA; BARROS; ASCENDINA (2014) inclui um módulo de monitoramento chamado Monitor of Driver/Device Comunication (MDDC) e um conjunto de máquinas de estados finitas (FSM) que são capazes de capturar erros no driver do dispositivo quando ele acessa registradores do dispositivo. Como pode ser visto na figura 1.3, o módulo MDDC captura a comunicação entre o controlador do dispositivo e o driver do mesmo. Dependendo do tipo de comunicação, a FSM correspondente verifica se houve um comportamento incorreto no dispositivo e no sistema.

Figura 1.3: Arquitetura do mecanismo de monitoramento



Fonte:MACIEIRA; BARROS; ASCENDINA (2014)

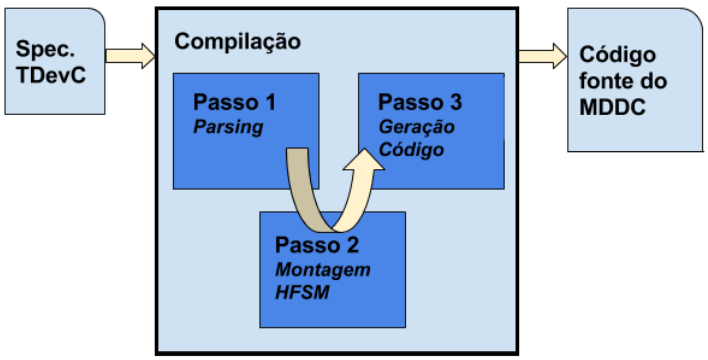
Esse módulo de monitoramento proposto é capaz de verificar se algumas propriedades como, se um registrador do dispositivo está sendo acessado de forma correta, bem como se registradores ou campos de registradores estão sendo acessados na ordem correta, estão ou não sendo violadas.

O módulo MDDC e o conjunto de FSMs são sintetizados a partir da especificação das características de um dispositivo na linguagem de especificação TDevC MACIEIRA; LISBOA; BARROS (2011). A linguagem TDevC é uma linguagem de domínio específico (DSL), que objetiva dar suporte à especificação do comportamento do protocolo de alto nível de comunicação entre o dispositivo e o driver, e de assertivas relacionadas a esse comportamento através de propriedades temporais (LTL).

A figura 1.4 mostra uma visão macro da abordagem apresentada em MACIEIRA; BARROS; ASCENDINA (2014). Após a especificação do modelo descrito em TDevC uma ferramenta chamada TDevCGen realiza a compilação desse modelo. Essa compilação, que pode ser dividida em 3 passos, inicia-se com parsing ou análise sintática da especificação em TDevC, que cria as listas de controle dos elementos estruturais e comportamentais da linguagem.

No passo 2, o modelo descrito em TDevC é transformado para um formato intermediário, montando na memória uma máquina de estados hierárquica. Cada estado dessa máquina, que possui como alfabeto expressões da lógica proposicional de primeira ordem, pode ter propriedades temporais comportamentais(LTL) que posteriormente serão validadas em cada um desses estados. No passo 3 o código-fonte do monitor é gerado em C++ ou SystemVerilog

Figura 1.4: Fluxo da abordagem apresentada em MACIEIRA; BARROS; ASCENDINA (2014)

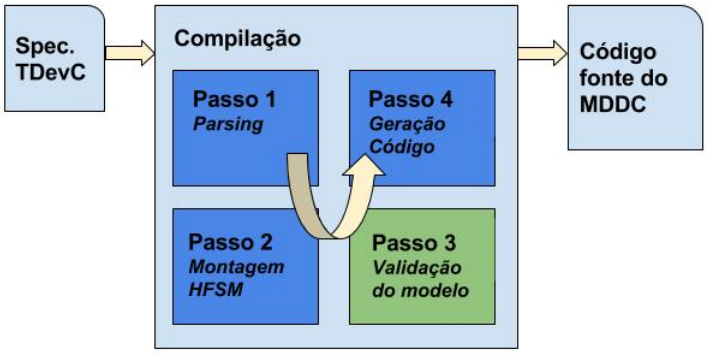


sintetizável.

Esta abordagem, no entanto, está sujeita a erros durante a descrição dos protocolos de comunicação de alto nível. Uma especificação em *TDevC* das características estruturais e comportamentais do *device driver* é feita manualmente a partir de documentos informais (*datasheets*), ou seja, a transcrição pode estar sujeita a falhas humanas de interpretação. Além disso, a própria documentação pode conter erros. Dessa forma, o presente trabalho tem como objetivo validar essa descrição e garantir que não haja inconsistências em relação ao protocolo descrito e nem contradições entre as propriedades temporais comportamentais.

Assim, a proposta consiste na realização de uma validação antes da geração do código fonte do monitor. A figura 1.5 mostra o fluxo da nova abordagem proposta, na qual pode ser visualizada a inclusão do passo 3, que trata da validação do modelo especificado antes da geração do monitor.

Figura 1.5: Fluxo da abordagem com a inserção da etapa de validação



Resumidamente temos o problema a ser resolvido e a solução proposta:

Problema: O monitor apresentado em MACIEIRA; BARROS; ASCENDINA (2014) é gerado a partir de uma máquina de estados hierárquica. Essa máquina porém, pode conter não-determinismos em seus estados e propriedades temporais comportamentais contraditórias. Caso o monitor seja gerado a partir de uma máquina de estados com um desses problemas, ele não será capaz de cumprir o objetivo ao qual foi proposto, que consiste em verificar se o device driver

que está sendo monitorado está implementado de acordo com a especificação. E nesse caso,seria preciso voltar para a etapa da especificação do modelo em TDevC e verificar se houve um erro na especificação ou um erro no modelo de referência.

Solução: A solução proposta consiste em realizar uma validação prévia da máquina de estados hierárquica antes da geração do monitor. Para essa validação foi utilizado um provador de teoremas de alto desempenho e as propriedades dos autômatos de Büchi.

Para facilitar a navegação e para o melhor entendimento, esse documento está dividido em capítulos e seções. No capítulo 2 são apresentados alguns conceitos básicos que são fundamentais para o entendimento da proposta. Em seguida, no capítulo 3, são apresentados alguns trabalhos relacionados à proposta. No capítulo 4, é apresentada a proposta deste trabalho e no capítulo 5 são apresentados os resultados dos experimentos realizados para a validação dessa proposta. Finalmente, no capítulo 6, uma conclusão é apresentada assim como propostas para os trabalhos futuros.

2

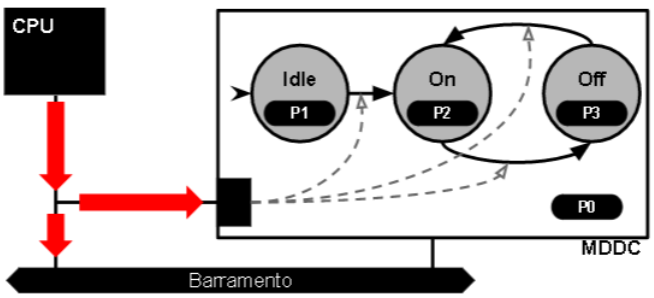
Fundamentação Teórica

O trabalho apresentado em MACIEIRA; BARROS; ASCENDINA (2014) propõe uma técnica para o monitoramento da comunicação entre *devices drivers* e seus respectivos dispositivos. A técnica utilizada para realizar esse monitoramento faz uso de uma linguagem para a descrição das propriedades a serem monitoradas e da arquitetura de um módulo de monitoramento sintetizado a partir das descrições dessas propriedades na linguagem de domínio específico TDevC.

Os módulos que realizam esse monitoramento, chamados de *monitors of driver/device commmunication* (MDDCs), ficam observando o meio de comunicação entre a CPU e os dispositivos durante a execução de uma plataforma embarcada. Eles verificam se as propriedades essenciais dos devices drivers estão sendo respeitadas com o objetivo de garantir que a plataforma execute de maneira correta e confiável.

Ao observar o meio de meio de comunicação, o MDDC captura os dados dos acessos entre a CPU e o dispositivo que está sob validação. A figura 2.1 ilustra o monitor recebendo os dados da comunicação. Ele usa esses dados para verificar se as propriedades foram ou não violadas. Para isso, o monitor faz uso de dois modelos de máquinas de estados: a máquina de estados finita hierárquica com dados (HFSM-D) e o autômato de Büchi (BA).

Figura 2.1: Exemplo simplificado do funcionamento interno de um MDDC



Na figura 2.1 é possível verificar também um exemplo de uma HFSM-D simplificada. Assim que a CPU acessa o dispositivo que está sob validação o monitor captura os dados desse acesso e o traduz em transições da HFSM-D. Baseado no exemplo mostrado, o estado inicial da

máquina de estados é o estado *Idle*. Dessa forma, como a HFSM-D é um modelo de referência do protocolo de comunicação entre o driver e o dispositivo sob monitoramento, espera-se que o dispositivo seja inicializado no estado *Idle*. Se o monitor receber como entrada mais um acesso válido, dependendo do acesso, esse pode resultar em uma outra transição da HFSM-D, como, no caso da figura 2.1, do estado *Idle* para o estado *On*.

É possível verificar também na figura 2.1 que cada estado da máquina contém um bloco representando o conjunto de propriedades que aquele estado deve respeitar. São eles: $P1$ para o estado *Idle*, $P2$ para o estado *On* e $P3$ para o estado *Off*. Existe também um bloco de propriedades ($P0$) fora dos estado, que são as propriedades globais, as quais devem ser respeitadas em todos os estados. Essas propriedades temporais comportamentais são descritas na linguagem TDevC usando notações da lógica temporal linear (LTL).

Esse monitor sintetizado pode estar sujeito, porém, a dois tipos de comportamentos indesejados:

1. Estados com a presença de não-determinismo: um estado não-determinístico é aquele no qual para cada estados e símbolos de entrada pode haver vários próximos estados possíveis.

2. Existência de propriedades temporais contraditórias: uma máquina de estados hierárquica pode ter várias sequências de execução. A sequência escolhida irá depender do comportamento do driver durante a execução. Cada linha de execução possui uma sequência de propriedades temporais que devem ser satisfeitas. Essas propriedades determinam o comportamento desejado. Porém, existe a possibilidade das propriedades se contradizerem em algum momento da execução.

Dessa forma, o objetivo deste trabalho consiste em validar a máquina de estados hierárquica para que o monitor não seja gerado com esses tipos de comportamentos. A técnica utilizada na validação faz uso de um provador de teoremas e das propriedades dos autômatos de Büchi.

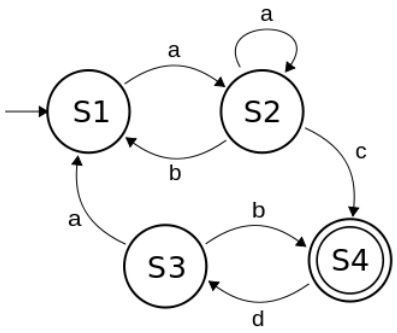
Para entender melhor a proposta, nas próximas seções serão explicados com mais detalhes a máquina de estados hierárquica, a partir da qual o monitor será gerado; a lógica temporal linear, que será a notação utilizada para descrever as propriedades temporais comportamentais do dispositivo; a DSL TDevC, que será utilizada para a especificação da máquina hierárquica juntamente com as propriedades temporais dos dispositivos; e por fim será apresentada uma visão geral dos provadores automáticos de teoremas, que serão utilizados na verificação de não-determinismos na máquina de estados hierárquica.

2.1 Máquina de Estados Hierárquica com dados

Uma máquina de estados finita (FSM - Finite State Machine), ou autômato finito, é um modelo matemático usado para representar programas de computadores ou circuitos lógicos. O

conceito pode ser expresso como uma máquina abstrata que deve estar em um dos seus estados em um conjunto finito de estados. Essa máquina deve estar em apenas um estado por vez, chamado de estado atual. Uma transição indica uma mudança de estado e é descrita por uma condição que precisa ser realizada para que a transição ocorra. Uma ação é a descrição de uma atividade que deve ser realizada em um determinado estado.

Figura 2.2: Exemplo de uma máquina de estados finita



A figura 2.2 mostra um exemplo de máquina de estados finita. Nela é possível verificar a presença do estado inicial S1, dos estados intermediários S2 e S3 e do estado final S4. As condições de transição são representadas pelas letras a, b, c e d.

Máquinas de estados finita podem modelar um grande número de problemas, entre os quais podemos citar a automação de um projeto eletrônico e protocolos de comunicação. Considerando o contexto desse trabalho, a máquina de estados pode representar o comportamento do dispositivo sob validação assim como o comportamento indesejado específico durante a execução da plataforma. É possível organizar uma FSM usando máquinas de estados finitas hierárquicas.

Uma HFSM-D é uma máquina de estados que representa o protocolo de comunicação entre dispositivos e cada estado está associado a um momento na execução desse protocolo. Cada estado da HFSM-D pode ter propriedades associadas, as quais serão traduzidas em autômatos de Büchi. Esta seção explicará em detalhes e formalmente a HFSM-D.

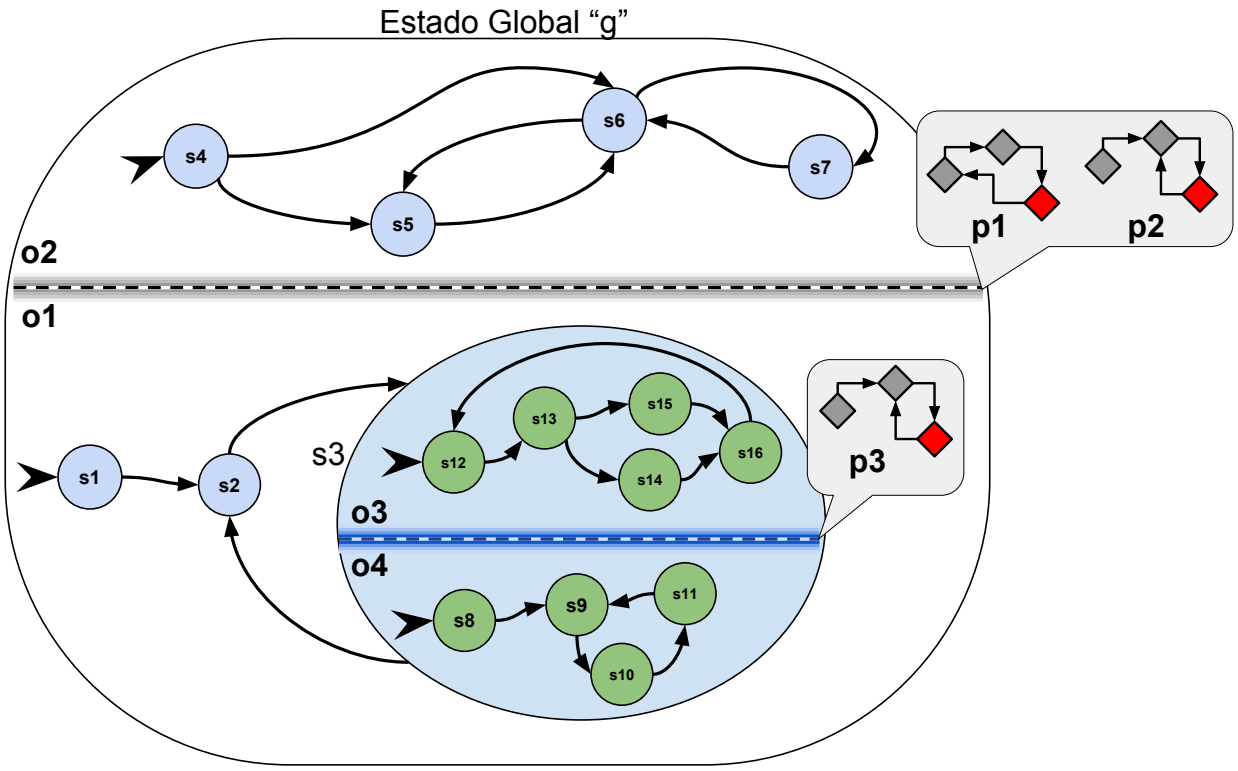
Uma HFSM-D é uma máquina de estados hierárquica, ou seja, ela pode conter uma ou mais sub-máquinas dentro de cada estado. A partir daí, surge o conceito de estados filhos que são aqueles pertencentes a uma sub-máquina de outro estado e estados irmãos, que são aqueles que possuem o mesmo pai. Assim, com base na figura 2.3 temos que os estados *s8* e *s12* são filhos do estado *s3*, sendo esse o pai dos estados *s8* e *s12*. Além disso, por terem o mesmo pai, os estados *s8* e *s12* são considerados irmãos.

O *Estado Global* é a raíz de uma HFSM-D. É considerado o único estado ancestral de todos os outros estados da máquina hierárquica e não possui um estado pai. Na figura 2.3 o estado global é o estado *g*. Imediatamente após cada estado pai podem existir duas ou mais sub-máquinas de estados paralelas e distintas que correspondem a linhas de execução simultâneas. Cada uma dessas sub-máquinas paralelas encontram-se em uma região denominada *Região*

Ortogonal. O conceito de região ortogonal foi definido baseado em extenções UML para o formalismo FSM tradicional, no qual é aplicada a operação de "*ou exclusivo*" para um estado, indicando que apenas um descendente daquele estado pode executar de cada vez.

Portanto, cada estado não-folha pode conter uma ou mais Regiões Ortogonais, em que cada uma delas possui uma máquina de estados distinta e de execução paralela.

Figura 2.3: Exemplo hipotético de uma HFSM-D



O exemplo mostrado na figura 2.3 contém quatro Regiões Ortogonais. As regiões *o1* e *o2* pertencem ao estado global *g* e as regiões *o3* e *o4* pertencem ao estado *s3*. As Regiões Ortogonais foram definidas com o objetivo de impedir diretamente na especificação a realização de transições entre sub-máquinas de estados irmãos. Cada máquina de estados existente em uma *Região Ortogonal* necessita de um *Estado Inicial*; na figura 2.3 esses estados são *s1*, *s4*, *s8* e *s12*.

As Regiões Ortogonais foram explicitamente definidas na linguagem TDevC com o objetivo de impedir diretamente na descrição dos modelos, que transições entre sub-máquinas de diferentes estados sejam realizadas. Como são linhas de execuções distintas, não pode haver junções nos fluxos de transições de diferentes sub-máquinas de estados concorrentes e nem em estados em diferentes níveis de hierarquia.

Cada estado pode conter um conjunto de propriedades temporais. Representadas pelos símbolos *p1*, *p2* e *p3*, essas propriedades possuem um escopo de abrangência limitado pelo estado na qual estão presentes. Por exemplo, como as propriedades *p1* e *p2* estão localizadas no estado global *g*, elas deverão ser satisfeitas em todos os estados da máquina. Já a propriedade *p3* está diretamente relacionada ao estado *s3* e indiretamente associada aos seus filhos: *s8*, *s9*, *s10*,

s11, *s12*, *s13*, *s14*, *s15* e *s16*. Dessa forma, a propriedade *p3* terá efeito apenas no estado *s3* e nos seus descendentes, não tendo efeito sobre os demais estados filhos do estado global *g*. Assim, um determinado estado possui as propriedades declaradas naquele estado e as propriedades de seus ascendentes hierárquicos.

De forma simplificada, tanto a HFSM-D quanto as máquinas que representam as propriedades temporais simbolizam *traces* de acessos realizados por um device driver ao seu respectivo dispositivo. Entretanto, os traces especificados na HFSM-D servem basicamente como um guia do protocolo. Cada acesso equivale a um evento que pode disparar uma transição entre estados na HFSM-D. Assim, cada estado desta máquina contém, além das transições para outros estados, uma transição para si conhecida como "else". Essa transição é realizada toda vez que um evento não for representado por nenhuma das transições existentes do estado em questão. Isso significa que, para aquele nível de abstração da especificação TDevC ou para aquela região ortogonal, aquele evento é indiferente para o protocolo.

Entretanto, ser indiferente para o protocolo ou para àquela região ortogonal não significa ser indiferente para a segurança e estabilidade da execução da plataforma. É justamente nesse ponto que traces representados pelos autômatos de Büchi são monitorados. Esses traces representam essencialmente propriedades da comunicação entre drivers e periféricos em pontos específicos do protocolo (estados da HFSM-D). Com essas duas formas de expressar traces, é possível separar claramente protocolos básicos de comunicação para o uso de um periférico de propriedades comportamentais da execução desta comunicação associados a pontos específicos desses protocolos.

2.1.1 Formalização da HFSM-D

Formalmente temos que uma HFSM-D é formada por um conjunto de estados S e sendo um estado $s \in S$, temos que s pode ser definido formalmente da seguinte forma:

Definição 1. *(Estado pertencente a HFSM-D)*

- l *é o nível de hierarquia do estado, onde* $l \in \mathbb{N}$;
- $a \cup \{\}$ *é o estado pai de* s*, tal que* $a \in S$*. Se* s *for o estado global da HFSM-D, seu pai* $\in \{\}$;
- D_s *é o conjunto de descendentes ou filhos do estado, tal que* $D_s \subset S$*. Se* s *for um estado folha,* $D_s = \{\}$;
- P_s *é o conjunto de propriedades temporais comportamentais que devem ser respeitadas pelo estado;*
- O_s *é o conjunto de regiões ortogonais do estado. Se* s *for um estado folha,* $O_s = \{\}$

Uma HFSM-D pode então ser definida da seguinte maneira:

Definição 2 (Definição da HFSM-D)**.** *Sendo H uma HFSM-D, tem-se que $H = \langle g, S, V, \Sigma, \delta, O, \phi, y \rangle$, onde:*

- *g é o estado global e pai de todos os outros estados da HFSM-D, onde $g \in S$;*
- *S é o conjunto de todos os estados da HFSM-D;*
- *V é o conjunto de variáveis da HFSM-D, onde $\forall v \in V, \exists k_v \in \mathbb{I}$. k_v representa o valor associado a esta variável v e $\mathbb{I}$ representa o conjunto dos números inteiros;*
- *Σ é o alfabeto de entrada da HFSM-D e σ é um expressão da lógica proposicional, onde $\sigma \in \Sigma$;*
- *δ é a função de transição da HFSM-D, onde $\delta(S \times \Sigma \times ver(\Sigma)) \rightarrow S$, tal que a função ver é a funcão de veracidade SMULLYAN (1995), onde $ver(X) \rightarrow \{t, f\}$ indica se a expressão X é verdadeira (t) ou falsa (f);*
- *ϕ é a função de definição dos valores k_v para $v \in V$, onde $\phi(S \times \nu(\Sigma) \times V) \rightarrow \mathbb{I}$.*
- *y define a granularidade ou unidade atômica de tempo utilizado na contagem de tempo-real, onde $y \in \mathbb{R}$, uma vez que a unidade de y é segundos (s). A cada intervalo de tempo y um evento de transição é disparado na HFSMD, ou seja, a função δ é invocada.*

A definição acima permite representar qualquer sub-máquina de H a partir da seguinte forma: sendo H_s uma sub-HFSMD de H a partir de um estado $s \in S$, tem-se que $H_s = \langle s, \{s\} \cup R_s, V, \Sigma, \delta, \phi, y \rangle$.

O alfabeto Σ da HFSM-D é formado por expressões da lógica proposicional de primeira ordem SMULLYAN (1995) cujas variáveis proposicionais são formadas a partir de proposições atômicas compostas por referências aos dados dos acessos realizados pela CPU a periféricos, valores de variáveis, estados correntes da própria HFSM-D e valores de temporizadores para propriedades de tempo-real.

Considerando π um *trace* ou caminho de execução e comparando a máquina de estados clássica com uma leitora de fita magnética idealizada por HOPCROFT; MOTWANI; ULLMAN (2006), cada elemento de π é entregue para a HFSM-D toda vez que a *cabeça de leitura* encontra seu próximo elemento na fita. O gatilho para que a leitora da fita faça rolar o rolo de leitura é um evento temporal de acesso ao dispositivo. Esses acessos são realizados através de escritas ou leituras em registradores do dispositivo.

Um evento de acesso $b \in \pi$ pode ser representado através de uma 3-túpla, no qual $b = \langle \{r, w\}, e, d \rangle$, onde:

- $\{r, w\}$ é o conjunto dos tipos do acesso. O tipo r representa uma leitura e o tipo w uma escrita.
- e representa o endereço acessado, onde $e \in \mathbb{N}$

- d representa o valor do dado vinculado ao acesso, onde $d \in \mathbb{I}$

Cada proposição atômica das expressões pertencentes a Σ quando aplicada a uma função de veracidade semelhante à função *ver* apresentada anteriormente deve ter como resposta um elemento do mesmo conjunto $\{t, f\}$. A importância de definir com mais detalhes as proposições atômicas reside no fato de ser uma informação extremamente necessária para definir os tipos dos dados extraídos de uma especificação TDevC. Dessa maneira, as proposições atômicas são definidas a seguir:

- $compS(S \times m_c) \rightarrow \{t, f\}$: A partir de um elemento do conjunto de estados S e um momento atual m_c, o resultado desta função é verdade quando o estado em questão é o um estado na qual a HFSM-D se encontra no momento atual m_c.

- $compT(B \times \{r, w\}) \rightarrow \{t, f\}$: A partir de um elemento do conjunto de acessos B e um elemento do conjunto de tipo de acesso $\{r, w\}$, o resultado desta função é verdade quando o acesso é do tipo informado.

- $compE(B \times \mathbb{N} \times \{=, \neq, \leq, <, \geq, >\}) \rightarrow \{t, f\}$: A partir de um elemento do conjunto de acessos B, um número pertencente ao conjunto dos $\mathbb{N}$ e um operador de comparação numérico, o resultado desta função é verdade quando o endereço do acesso e o valor do número natural informado respeitam o operador de comparação.

- $compD(B \times \mathbb{I} \times \{=, \neq, \leq, <, \geq, >\}) \rightarrow \{t, f\}$: A partir de um elemento do conjunto de acessos B, um número pertencente ao conjunto dos $\mathbb{I}$ e um operador de comparação numérico, o resultado desta função é verdade quando o valor associado ao acesso e o valor do número inteiro informado respeitam o operador de comparação.

- $compV(V \times \mathbb{I} \times \{=, \neq, \leq, <, \geq, >\}) \rightarrow \{t, f\}$: A partir de um elemento do conjunto de variáveis V, um número pertencente ao conjunto dos $\mathbb{I}$ e um operador de comparação numérico, o resultado desta função é verdade quando o valor k_v associado à variável e o valor do número inteiro informado respeitam o operador de comparação.

Respeitando o formalismo da lógica de primeira ordem apresentada por SMULLYAN (1995), temos que a relação entre as proposições atômicas para a formação de fórmulas mais complexas se dá através dos operadores clássicos da lógica proposicional contidos no conjunto Ω. São eles: conjunção ($\wedge$), disjunção ($\vee$), negação (!, $\neg$ ou $\sim$), implicação ($\Rightarrow$) e equivalência ($\Leftrightarrow$).

2.2 Lógica Temporal Linear

A especificação de um sistema pode ser composta por um conjunto de propriedades temporais que o definem. Introduzida por Pnueli em 1977 PNUELI (1977), Lógica Temporal

Linear LTL é uma lógica para a especificação de propriedades temporais de sistemas reativos. LTL provê um formalismo para a especificação de propriedades de sequências de execução de um sistema com o objetivo de expressar suas propriedades comportamentais J.KATOEN; BAIER (2008).

Embora o termo *temporal* sugira uma relação com o comportamento em tempo real de um sistema reativo, isso só é verdade em um sentido abstrato. Uma lógica temporal permite a especificação da ordem relativa de eventos. Alguns exemplos suportados são "o carro para, assim que o condutor empurra o freio" ou "a mensagem é recebida, depois de ter sido enviada". No entanto, não suporta qualquer meio para se referir à data precisa de eventos, por exemplo, eventos do tipo "o sistema do carro tem um atraso mínimo de 3μs entre a travagem e a parada real do veículo".

Em termos de transições do sistema, nem a duração de atingir um estado específico e nem a quantidade de tempo gasto em um determinado estado podem ser especificados usando a lógica temporal. Em vez disso, essa lógica pode ser usada para especificar a ordem em que as etiquetas de estado ocorrem durante uma execução ou para verificar que certas etiquetas de estado ocorrem ou não infinitamente na execução de um sistema J.KATOEN; BAIER (2008).

As próximas seções descreverão a sintaxe e a semântica de LTL.

2.2.1 Sintaxe

As sentenças atômicas são os elementos básicos que constituem a sintaxe da LTL. São proposições declarativas que podem assumir o valor verdadeiro ou falso e que não podem ser divididas em outras sentenças mais simples. Por exemplo, "O computador é branco" é uma sentença atômica em linguagem natural. A seguir tem-se a definição formal da sintaxe da LTL ROZIER (2010):

Definição 3 (Definição da sintaxe da LTL)**.** *Seja PA um conjunto de proposições atômicas, então:*

- *Cada proposição atômica $p \in PA$ é uma fórmula LTL;*
- *Se ϕ é uma fórmula, então $\neg\phi$ é uma fórmula;*
- *Se ϕ e ψ são fórmulas, então $\phi \vee \psi$ é uma fórmula;*
- *Se ϕ é uma fórmula, então $\bigcirc\phi$ é uma fórmula (lê-se "próximo fi");*
- *Se ϕ e ψ são fórmulas, então $\phi\ U\ \psi$ é uma fórmula (lê-se "fi até que psi");*
- *Nada mais é uma fórmula.*

Os operadores $\neg$ e $\vee$ são respectivamente de negação e disjunção. É a partir deles que os operadores booleanos $\wedge$ (conjunção), $\Rightarrow$ (implicação) e $\Leftrightarrow$ (equivalência), assim como a definição de *true* e *false* são derivados:

- $\phi \wedge \psi \equiv \neg(\neg\phi \vee \neg\psi)$
- $\phi \Rightarrow \psi \equiv \neg\phi \vee \psi$
- $\phi \Leftrightarrow \psi \equiv (\phi \Rightarrow \psi) \wedge (\phi \Rightarrow \psi)$
- $true \equiv \phi \vee \neg\psi$
- $false \equiv \neg true$

O operador $\bigcirc$ é um operador unário e requer apenas uma fórmula LTL como argumento. Uma fórmula $\bigcirc\phi$ é verdadeira no estado atual somente se ϕ for verdaira no próximo estado. O operador U é binário e requer duas fórmulas LTL como argumento. A fórmula $\phi\ U\ \psi$ é verdadeira no estado atual se ϕ é válida ao longo de toda uma sequência de estados consecultivos até a ocorrência de ψ. Os operadores temporais $\Box$ (lê-se "sempre"ou "globalmente") e $\Diamond$ (lê-se "futuramente") são definidos a partir dos operadores definidos anteriormente:

$$\Diamond\phi \equiv true U\ \phi$$
$$\Box\phi \equiv \neg\Diamond\neg\phi$$

$\Diamond\phi$ garante que ϕ será eventualmente verdadeira no futuro e $\Box\phi$ é satisfatível se e somente se $\neg\phi$ não for verdadeira em nenhum momento.

2.2.2 Semântica

Um modelo é uma função M:$\mathbb{N}_0 \rightarrow 2^{PA}$, onde $\mathbb{N}_0$ é o conjunto 0,1,2,... dos números naturais. Em outras palavras, um modelo é uma sequência infinita P_0P_1... de subconjuntos de Proposições Atômicas (PA). A função M descreve como a verdade de proposições atômicas muda a medida que o tempo avança.

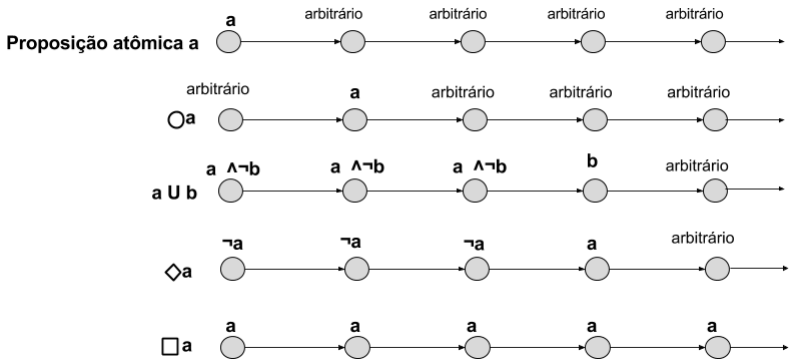
Escreve-se $M,i \models \alpha$ para denotar que "α é verdadeira no instante de tempo i no modelo M". Essa noção é definida indutivamente de acordo com a estrutura de α ROZIER; VARDI (2007).

- $M,i \models$ p, onde p $\in$ PA, sse p $\in M(i)$.
- $M,i \models \neg\alpha$ sse $M,i \not\models \alpha$.
- $M,i \models \alpha \vee \beta$ sse $M,i \models \alpha$ ou $M,i \models \beta$.
- $M,i \models \bigcirc\alpha$ sse $M,i+1 \models \alpha$.
- $M,i \models \alpha\ U\ \beta$ sse existe $k \geq i$ tal que $M,k \models \beta$ e para todo j tal que $i \leq j < k$, $M,j \models \alpha$.

A fórmula α é dita ser satisfatível se existe um modelo M e um instante i tal que $M,i \models \alpha$.

Com relação aos conectivos derivados $\Diamond$ (futuramente) e $\Box$ (globalmente) temos:

Figura 2.4: Semântica dos operadores temporais



Fonte:J.KATOEN; BAIER (2008)

- $M,i \models \Diamond\alpha$ sse existe $k \geq i$ tal que $M,k \models \alpha$
- $M,i \models \Box\alpha$ sse para todo $k \geq i$ tal que $M,k \models \alpha$

A figura 2.4 faz um resumo da semântica dos operadores temporais.

2.2.3 Linguagem ω-regular

As linguagens ω-regular são uma classe das ω-linguagens (linguagem formada por palavras de comprimento infinito de símbolos) que generalizam a definição de linguagens regulares para palavras de comprimento infinito.

Definição 4 (Linguagem ω-regular)**.** *Uma ω-linguagem é ω-regular se tem a forma:*

- *A^ω em que A é uma linguagem regular não vazia que não contém a palavra vazia ε. Os elementos de A^ω são obtidos concatenando as palavras de A infinitas vezes;*
- *AB, em que A é uma linguagem regular e B é uma linguagem ω-regular;*
- *$A \cup B$ em que A e B são linguagens ω-regular.*

O conjunto das linguagens ω-regulares é fechado sobre as operações de união, interseção e complementação, ou seja, o resultado de qualquer uma dessas operações sobre linguagens ω-regulares é também uma linguagem ω-regular MUKUND (1997).

As linguagens ω-regulares corresponde ao conjunto de linguagens aceitas por um autômato de Büchi, como mostra o seguinte teorema.

Teorema 1. *Uma linguagem ω-regular é reconhecida por um autômato de Büchi se e somente se for uma linguagem ω-regular.*

2.2.4 Autômatos de Büchi

Há uma ligação íntima entre os modelos das fórmulas LTL e as linguagens de palavras infinitas: os modelos de uma fórmula LTL constituem uma linguagem ω-regular sobre um alfabeto apropriado. Beneficiando-se do fato que toda LTL possui um autômato de Büchi equivalente, o problema de satisfatibilidade para fórmulas LTL se reduz a verificar se o conjunto das linguagens ω-regular é ou não vazio.

Um autômato de Büchi (BA) é uma extensão de um autômato finito (FSM). Eles possuem a mesma sintaxe, porém, a semântica, designada pela caracterização da aceitação de uma palavra, é diferente, uma vez que autômatos de Büchi lidam com palavras infinitas.

Definição 5 (Autômato de Büchi (não-determinístico))**.** *Um autômato de Büchi A pode ser formalmente definido como uma 5-upla (Q,*Σ*,*δ*,*q_0*,F) onde:*

- *Q é um conjunto finito e representa o conjunto não-vazio de estados de A*
- Σ *é o conjunto finito de símbolos do alfabeto de A*
- δ*: Q x* $\Sigma \rightarrow$ *Q é a função de transição de A*
- q_0 *é um elemento de Q chamado de estado inicial de A*
- $F \subseteq Q$ *é o conjunto que representa o estado de aceitação. A aceita exatamente as execuções que passam infinitamente por pelo menos um estado de aceitação.*

Uma trajetória em um BA A para uma palavra infinita $\alpha = a_1a_2a_3...$ é uma sequência inifinita $\rho = (r_0, r_1, r_2,...)$ de estados de A que satisfaz as seguintes condições:

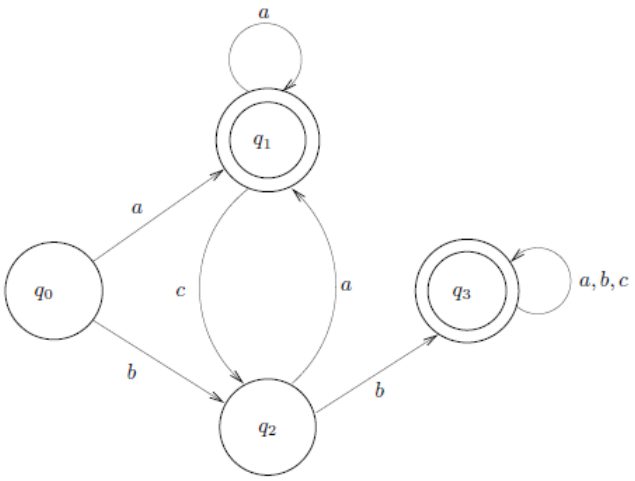
$$\text{(i) } r_0 \in q_0$$
$$\text{(ii) } r_i \in \delta(r_{i-1}, a_i), \forall i > 0$$

Seja A = (Q,Σ,δ,q_0,F) um BA e α uma palavra infinita sobre o alfabeto Σ, A aceita α se e somente se existe uma trajetória $\rho = (r_0, r_1, ..., r_n)$ de α em A tal que $r_n \in$ F, ou seja, se há uma trajetória da palavra no autômato que termina em algum estado final de A. Dizemos que A não aceita α se não existe trajetória de α sobre A que termina em um estado final.

A linguagem aceita por um autômato de Büchi A é definida como o conjunto de palavras, denotada por L(A). Por exemplo, considerando o BA A = (Q,Σ,δ,q_0,F), tal que:

- $Q = q_0, q_1, q_2, q_3$
- $\Sigma = a, b, c$
- $\delta = (q_0, a, q_1), (q_0, b, q_2), (q_1, a, q_1), (q_1, c, q_2), (q_2, a, q_1), (q_2, b, q_3), (q_3, a, q_3), (q_3, b, q_3), (q_3, c, q_3)$
- q_0: é um elemento de Q chamado de estado inicial de A
- $F = q_1, q_3$

Figura 2.5: Exemplo de um Autômato de Büchi



A figura 2.5 mostra a representação gráfica de A, em que os estados com dois círculos concentrícos (q_1 e q_3) são os estados finais.

Considerando a palavra infinita:

$$\alpha = acacacacaca...$$

tem-se que ela é aceita por A, uma vez que passa infinitas vezes pelo estado final q_1. Por outro lado, a palavra:

$$\beta = \text{baccccccc...}$$

não é aceita por A, pois não existe trajetória em A que passe infinitas vezes por algum estado final.

2.3 A Linguagem TDevC

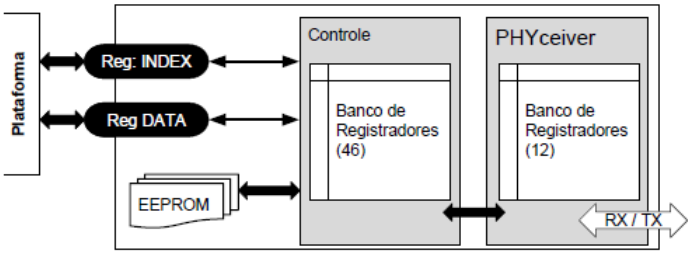
Inicialmente proposta em LISBOA et al. (2009), a TDevC é uma linguagem de domínio específico que permite especificar elementos estruturais, protocolos de acesso à memória de dispositivos e comportamentos ideais para o uso dos dispositivos.

Para uma melhor compreensão de uma especificação nessa linguagem, primeiramente será explicado em detalhes a interface de comunicação e logo em seguida o protocolo de comunicação. Além disso, para esclarecer a explicação, todos os exemplos da sintaxe da linguagem utilizados nas explicações foram retirados da especificação real de um controlador de *Ethernet DM9000A*.

A Ethernet DM9000A DAVICOM (2006) consiste de um dispositivo de rede cabeada que implementa parte da pilha de protocolos do modelo ISO/OSI. A figura 2.6 mostra um diagrama de blocos simplificado da arquitetura da DM9000A. No diagrama verifica-se que o dispositivo é composto por dois bancos de registradores: um na unidade de controle, contendo 46 registradores, e outro na unidade da camada física (*PHYceiver*), contendo 12 registradores.

A unidade de controle é responsável pelo controle de todas as funcionalidades das

Figura 2.6: Diagrama de blocos simplificado da Ethernet DM9000A



camadas acima da camada física, enquanto que a unidade *PHYceiver* é responsável pelo controle das funcionalidades da camada física do protocolo de rede.

A interface de comunicação desse dispositivo com a CPU contém apenas dois registradores de 8 bits cada: *INDEX* e *DATA*. Estes registradores são utilizados como porta de entrada para o acesso ao banco de registradores da unidade de controle.

Os registradores internos, tanto da unidade de controle como da unidade *PHYceiver*, são identificados através de um endereço. O acesso externo a esses registradores é realizado através dos dois registradores da interface do dispositivo, colocando no registrador *INDEX* o endereço do registrador interno que deseja-se acessar (leitura/escrita) e o dado lido ou escrito no registrador *DATA*. Porém, o banco de registradores da unidade *PHYceiver* e a memória EEPROM não podem ser acessados diretamente a partir dos registradores da interface. Para realizar esse acesso é necessário utilizar registradores específicos do banco de registradores da unidade de controle.

2.3.1 Definição da interface de comunicação na Linguagem TDevC

A especificação da interface de comunicação do dispositivo na linguagem TDevC é composta pela especificação dos registradores do dispositivo, dos formatos desses registradores e dos padrões de dados através das construções *register*, *format* e *pattern*, respectivamente. A figura 2.7 mostra um exemplo dessas construções.

A especificação *TDevC* de um dispositivo inicia-se com a palavra reservada **device** (linha 1), seguida pelo nome do dispositivo cujo modelo está sendo especificado, que no exemplo mostrado trata-se do Controlador *Ethernet DM9000A*.

Os padrões permitem especificar formatos numéricos de máscaras, tornando a descrição do comportamento mais clara e menos propensa a erros. A declaração de padrões na linguagem *TDevC* é realizada através da palavra reservada **pattern** (linha 2). Como pode ser visto na linha 2 da figura 2.7, a construção é bastante parecida com a declaração de variáveis constantes de linguagens de programação, porém, além de valores fixos é possível definir formatos fixos de dados. No exemplo mostrado, o padrão especificado RXNOERROR define o formato do dado que deve ser lido quando houver um erro na transmissão de um pacote de dados, independente da origem deste dado. Na linha 2 do exemplo mostrado, a construção **mask** define a máscara

```
1  device (dm9000a) {
2    pattern RXNOERROR = mask(....0000);
3
4    format physicalAddrfmt {
5      RW PAB[7:0];
6    }
7
8    external register indexReg(0x00) alias INDEXREG {
9          RW INDEX [15:0];
10     }
11
12   internal IntRegsProt register networkStatusReg(0x00) alias NSR {
13     READ SPEED [7];
14     READ LINKST [6];
15     RW WAKEST [5];
16     reserved[4];
17     RW TX2END [3];
18     RW TX1END [2];
19     READ RXOV [1];
20     reserved[0];
21   }
22
23   internal IntRegsProt register CHIPRevision(0x2C) alias CHIPR {
24         READ CHIPR[7:0];
25     }
26
27   internal IntRegsProt register phyAddrReg5(0x15) alias PAR5 = physicalAddrfmt;
28 ...
```

Figura 2.7: Exemplo de uma descrição da interface de comunicação na linguagem TDevC

que um dado lido ou escrito deve respeitar ao ser comparado com esse padrão. A máscara é definida com a composição de valores do tipo zero (0), tipo um (1) ou tipo ponto (.). O valor do tipo zero define que o bit naquela posição deve obrigatoriamente possuir o valor igual zero (0) e do tipo um o valor igual a um (1). Já valores do tipo ponto indicam que o bit naquela posição pode possuir o valor zero (0) ou um (1). No caso do exemplo apresentado, o dado deve conter os quatro primeiros bits iguais a zero (0), considerando que o notação *little-endian* (o byte de ordem mais baixa do dado está armazenado no endereço de memória mais baixo) está sendo utilizada.

A construção **register** representa a declaração de um registrador real do dispositivo modelado. Os registradores podem ser declarados explicitamente ou através da construção **format**. Quando declarados explicitamente, os registradores possuem a sua visibilidade (interna ou externa), seguida do nome e do endereço físico e, opcionalmente, de seu apelido ou nome fantasia (*alias*), além disso, possuem a descrição dos campos do registrador, juntamente com a permissão de acesso de cada um desses campos.

Como dito anteriormente, os registradores possuem dois tipos de visibilidade: externa, representados através da palavra reservada **external**, e interna, representados através da palavra reservada **internal**. Os registradores externos são àqueles mapeados na faixa de endereçamento da plataforma, ou seja, todos os registradores que podem ser acessados diretamente pelos componentes mestres do sistema durante a comunicação. Usando como exemplo o dispositivo Ethernet, os registradores externos são o *INDEX* (construção apresentada na linha 8 da figura 2.7) e o *DATA*.

Os registradores internos não podem ser acessados diretamente através da faixa de endereços do sistema. Esses registradores são acessados através de um protocolo de acesso que faz uso dos registradores externos. Geralmente esses protocolos são implementados nas camadas de software dependente de hardware (HdS). Na figura 2.7, linhas 12 e 23, tem-se a declaração de dois registradores internos que utilizam um protocolo de acesso chamado *IntRegsProt*. A especificação dos protocolos será detalhada na seção 2.3.2.

É importante destacar que os registradores internos e externos possuem escopo de endereçamento diferente. Os registradores externos possuem seus endereços vinculados e traduzidos diretamente na plataforma alvo. Tomando como exemplo o registrador externo *indexReg*, percebe-se que seu endereço no dispositivo é $0x00$. Porém, caso o periférico em questão encontre-se na faixa de endereçamento $0x00A - 0x00E$ do sistema, o endereço relativo $0x00$ do registrador *indexReg* é traduzido para o endereço absoluto $0x00A$ da plataforma.

Os registradores internos, por sua vez, são relativos ao protocolo de acesso. Como cada protocolo definido carrega consigo um escopo diferenciado de endereçamento, é possível que registradores internos com diferentes protocolos e os registradores externos compartilhem o mesmo valor numérico de endereços. Um exemplo disso pode ser visto na figura 2.7-linhas 8 e 12.

O atributo não obrigatório **alias** define um nome fantasia ou apelido para o registrador, com o objetivo de permitir uma referência simplificada.

Os campos dos registradores, Figura 2.7-linhas 5, 9, 13 à 20 e 24, são subdivisões lógicas descritas nos *datasheets* (documentações oficiais) dos dispositivos. Geralmente cada subdivisão tem uma função específica. Além disso, os valores atribuídos a esses campos podem acarretar em mudanças de comportamento no dispositivo.

Caso o *datasheet* informe que o campo é reservado (podendo ser de uso interno, não estável ou ainda não definido pelo fabricante) usa-se na declaração desse campo reservado a palavra **reserved**. Nas linhas 16 e 17 da figura 2.7 é possível verificar essa construção.

Quando os campos são válidos para uso, a declaração requer três atributos obrigatórios e permite dois atributos opcionais. Com relação aos atributos obrigatórios, temos a permissão de acesso, podendo ter acessos somente para escrita (*WRITE*), somente para leitura (*READ*) ou para escrita e leitura (*RW*); o outro atributo é o nome do campo, o qual é usado na definição do protocolo da linguagem TDevC e é referenciado através do nome ou *alias* do registrador seguido de um "."acompanhado do nome do campo; o terceiro atributo obrigatório refere-se ao campo correspondente do registrador.

A declaração dos formatos é bem parecida com a declaração dos próprios registradores, diferenciando-se apenas pelo fato dos formatos não possuirem visibilidade, endereçamento e nome fantasia associados. Justifica-se essa declaração pelo fato desses atributos fazerem sentido apenas quando estão vinculados a registradores reais. Essa construção é bastante utilizada quando se tem um conjunto de registradores com os mesmos campos, bastando apenas definir o formato e depois atribuir esse formato aos diversos registradores. Na linha 4 tem-se a declaração de um

formato e na linha 27, a atribuição desse formato ao registrador *phyAddrReg5*.

2.3.2 Definição do protocolo de comunicação na Linguagem TDevC

A definição do Protocolo de Comunicação na linguagem *TDevC* é composta por construções que usam sintaxes específicas para a declaração de protocolos de acesso a registradores internos, declarações de variáveis de contexto e a declaração da máquina de estados hierárquica HFSM-D juntamente com suas atribuições de dados e propriedades comportamentais. A seguir, essas construções serão detalhadas, exemplificando-as com a especificação do controlador *Ethernet DM9000A*.

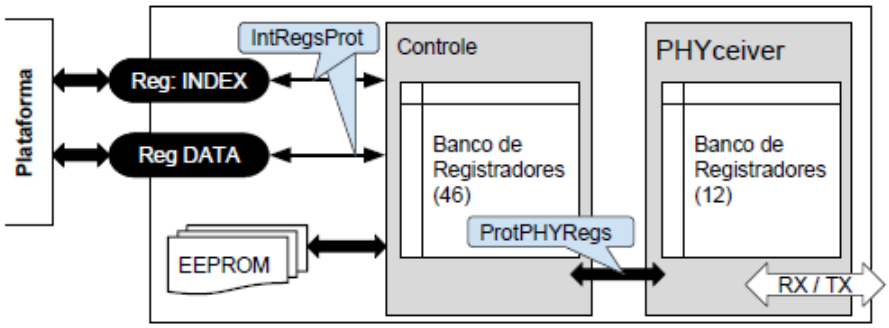
Como já mencionado na seção anterior, os protocolos são utilizados para definir os procedimentos de acesso aos registradores internos partindo de acessos a registradores externos. As linhas 1 e 12 da figura 2.8 mostram exemplos da especificação de dois protocolos, os quais são definidos para dar acesso aos registradores tanto da unidade de controle quanto da *PHYceiver* do controlador Ethernet. A figura 2.9 mostra onde esses protocolos estão inseridos no diagrama de blocos.

```
1  protocol IntRegsProt {
2      address: INDEXREG(0X00);
3      data: DATAREG;
4      readingtrigger{
5          read(DATAREG);
6      }
7      writingtrigger{
8          write{DATAREG};
9      }
10 }
11
12 protocol ProtPHYRegs{
13     address: EPAR.REGADDR(0X00);
14     data: {EPDR.EE_PHY_H;EPDR.EE_PHY_L}
15     readingtrigger{
16         write(EPCR) = 0x0C;
17     }
18     writingtrigger{
19         write(EPCR) = 0x0A;
20     }
21 }
```

Figura 2.8: Exemplo de declaração de protocolos de acesso a registradores internos na linguagem TDevC

O bloco da especificação do protocolo é iniciado pela palavra reservada **protocol**, seguida por um nome identificador do protocolo. Dentro do bloco, especifica-se primeiramente os registradores ou campos de registradores que serão utilizados para gravar o endereço do registrador interno e o dado que será lido ou escrito, as palavras reservadas utilizadas para tal fim são **address** e **data**, respectivamente. Na figura 2.8-linhas 2 e 3, por exemplo, o protocolo *IntRegsProt* define que o registrador *INDEXREG* terá o endereço do registrador interno que será acessado e no registrador *DATAREG* será armazenado o dado lido ou escrito.

Figura 2.9: Diagrama de blocos simplificado da Ethernet DM9000A com referência aos protocolos entre bancos de registradores



Ainda dentro do bloco, é especificado um gatilho que indica quando o dado está pronto para ser lido ou escrito. Para isso, as construções *readingtrigger* e *writingtrigger* foram definidas. Usando o exemplo da figura 2.8-linhas 15 a 20, a especificação do *Ethernet DM9000A* determina que para haver a leitura e a escrita, é preciso primeiramente escrever os valores $0x0C$ e $0x0A$ no registrador *EPCR*, respectivamente, informando que o endereço e o dado já foram configurados para realizar a leitura ou escrita.

Variáveis são utilizadas quando se deseja adicionar algum contexto durante a validação. Toda variável possui um escopo global e as atribuições feitas a elas ocorrem durante as transições da HFSM-D ou no momento da sua declaração. A linha de 1 da figura 2.10 mostram um exemplo da declarações de uma variável.

```
1  var t1 = 1;
2
3  globalstate {
4      orthoregion send_data {
5          initialstate WAIT_READY {
6              addexitpoint(SEND_DATA){write(TCR.TXREQ) == 0}
7              addexitpoint(SEND_DATA){write(CHIPR) == RXNOERROR}
8              addexitpoint(SEND_DATA){write(CHIPR) == 32}
9          }
10         state SEND_DATA {
11             addexitpoint(SEND_DATA) {
12                 read(NSR.TX1END) == 1 || read(NSR.TX2END) == 1
13             }
14             addexitpoint(WAIT_READY){read(NSR) == t1}
15         }
16     }
17 ...
```

Figura 2.10: Exemplo da descrição de um protocolo de comunicação na linguagem TDevC

A especificação da máquina de estados hierárquica inicia-se com a palavra reservada **globalstate**, que indica o estado global, único e pai de todos os demais estados. O estado global representa o primeiro nível (nível raiz) da HFSM-D. A linha 3 da figura 2.10 mostra o início do bloco do estado global e, consequentemente da HFSM-D.

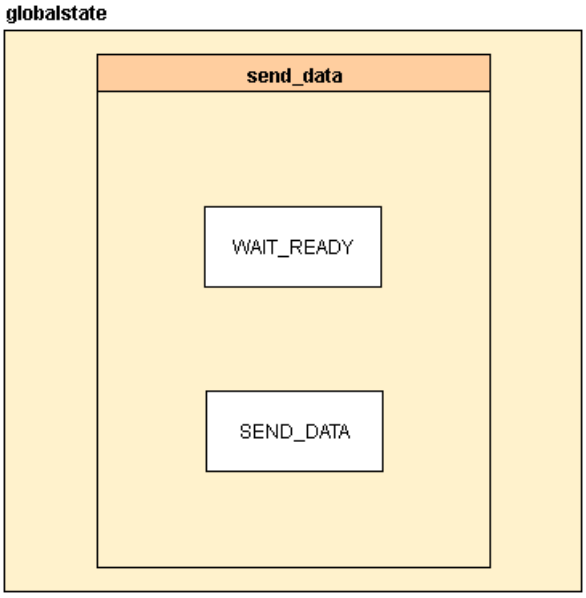
Respeitando a definição da HFSM-D dada na seção 2.1, o estado global, assim como

qualquer outro estado da máquina hierárquica, pode ser segmentado em regiões ortogonais através de sub-blocos iniciados com a palavra reservada **orthoregion**. Cada região ortogonal representa linhas de execuções diferentes dentro do estado ao qual pertence. A região ortogonal contém, também, em sua sintaxe a declaração de sub-estados, permitindo dessa forma a construção de uma máquina hierárquica.

Toda região ortogonal tem obrigatoriamente um único estado inicial declarado através da palavra reservada **initialstate**. Todos os demais estados, considerados intermediários de acordo com a definição da máquina hierárquica, são opcionais e declarados através da palavra reservada **state**.

O exemplo mostrado na figura 2.10 contém uma região ortogonal chamada *send_data* na linha 4. O seu estado inicial, declarado na linha 5, é o *WAIT_READY*, e o seu único estado intermediário é o *SEND_DATA*, na linha 10. A figura 2.11 mostra uma visão gráfica da máquina de estados hierárquica apresentada até agora.

Figura 2.11: HFSM-D com o estado global, região ortogonal e os estados

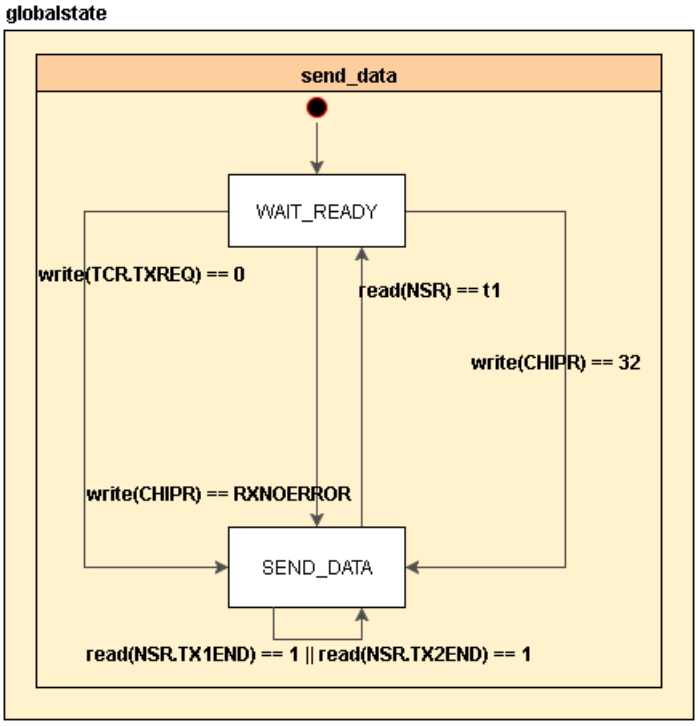


Dentro de cada bloco de definição de estado, dois atributos são responsáveis por especificar as transições entre os estados, são eles o **exitpoint** e o **entrypoint**. O bloco *exitpoint* define transições de saída e o *entrypoint* transições de entrada.

As transições de saída são sempre especificadas nos blocos que originam a transição, sendo disparadas por expressões lógicas, conforme a definição da função de transição da HFSM-D dada na seção 2.1. Nas linhas de 9 a 11 e de 14 a 17, tem-se exemplos de *exitpoints* que são adicionados através da palavra reservada **addexitpoint**. No exemplo, o estado inicial *WAIT_READY* possui três transições de saída para o estado *SEND_DATA*. Essas transições serão disparadas quando a expressão lógica especificada dentro do bloco for valorada como verdadeira. A primeira transição $write(TCR.TXREQ) == 0$, por exemplo, será verdadeira quando houver uma escrita no campo *TXREQ* do registrador *TCR* e o valor escrito for zero (0). Já a segunda

transição será disparada quando houver uma escrita no registrador *CHIPR* e o valor escrito for compatível com o padrão *RXNOERROR*. A figura 2.12 mostra uma visão gráfica da máquina com a adição das transições de saída de cada estado.

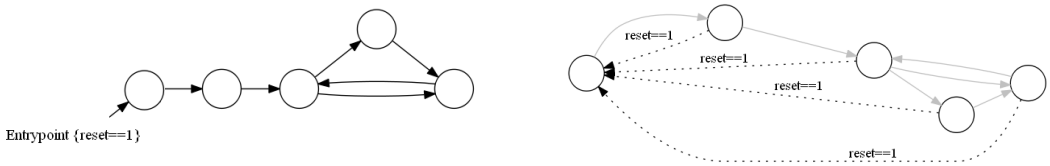
Figura 2.12: HFSM-D com o estado global, região ortogonal, os estados e suas transições



A TDevC também oferece as estruturas denominadas *entrypoints*. Os *entrypoints* foram criados para deixar mais transparente a ideia de transições comuns a todos os estados de uma sub-máquina hierárquica, uma vez que todos os *entrypoints* são convertidos em *exitpoints*. Ou seja, dentro de uma determinada região ortogonal, caso um estado possua um *entrypoint*, durante a etapa de síntese, esse *entrypoint* será transformado em um *exitpoint* em cada um dos outros estados da região ortogonal. A figura 2.13 mostra um exemplo dessa construção. A parte *a* da figura mostra um *entrypoint* entrando no primeiro estado. Esse *entrypoint* será transformado em um *exitpoint* saindo de cada um dos outros estados, como mostra a parte *b* da figura.

Para a especificação das propriedades que deverão ser validadas durante a execução do sistema, é utilizado a construção **addproperty**. As propriedades podem ser de dois tipos: críticas ou informativas. As propriedades críticas, declaradas com a palavra reservada **critical**,

Figura 2.13: Exemplo do transformação de um *entrypoint* em *exitpoints*



(a) Entrypoint (b) Entrypoint transformado em exitpoint

quando violadas interrompem imediatamente o monitoramento, visto que, após a violação, não há mais nenhuma garantia sobre o estado atual que o dispositivo se encontra. Já as propriedades informativas, declaradas com a palavra reservada **info**, não causam dano direto ao protocolo e à comunicação entre os periféricos. Nas linhas 1 e 7 da figura 2.14 tem-se a declaração de propriedades críticas e na linha 4 de uma propriedade informativa.

```
1 addproperty(critical) UndefinedOperMode {
2     ltlf([](~UNDEF_OPER))
3 }
4 addproperty(info) LenBeforeSend {
5     ltlf(~TXSDPKGSDING U TXWRPKGLEN)
6 }
7 addproperty(critical) NeverLenAndSend {
8     ltlf([]( ~(TXSDPKGSDING && TXWRPKGLEN)))
9 }
```

Figura 2.14: Exemplo da declaração de propriedades em estados na linguagem TDevC

As três propriedades mostradas na figura 2.14 pertencem ao estado chamado *PHYUP* da Ethernet DM9000A. Como já mencionado anteriormente, esse estado é alcançado quando o dispositivo está pronto para transmitir e receber pacotes de rede. Um passo prévio para deixar o dispositivo pronto é a escolha do modo de operação, que pode ser de 8 ou 16 bits. A escolha do modo de operação é realizada no estado *UNDEF_OPER*.

Assim, com o auxílio da explicação dos operadores da lógica temporal linear na Seção 2.2, é possível interpretar a primeira propriedade mostrada na linha 2. A propriedade $[](\sim UNDEF\_OPER)$ indica que sempre (operador temporal []) *UNDEF_OPER* deve ser negado (operador lógico $\sim$), ou seja, o estado *UNDEF_OPER* não pode ser um estado corrente em nenhuma outra região ortogonal, pois é necessário saber previamente o modo de operação, e não alterá-lo, durante a transmissão e no recebimento de pacotes de rede.

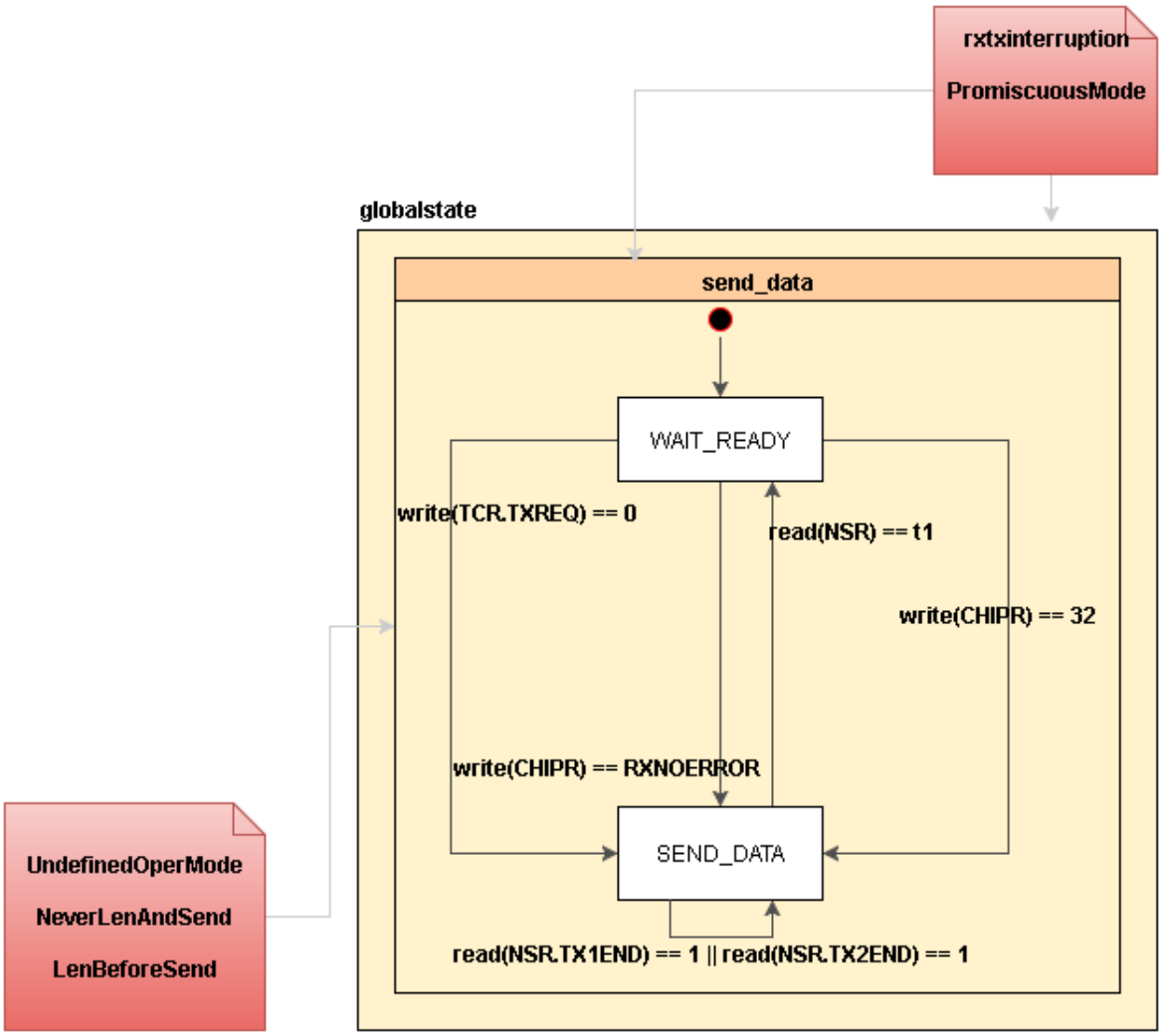
Como já explicado na seção 2.1, as propriedades possuem sua abragência limitada ao estado no qual foram declaradas e aos seus descendentes hierárquicos. Logo, propriedades declaradas no estado global possuem abragência em todos os estados da máquina hierárquica, enquanto que as propriedades declaradas no estado *PHYUP*, por exemplo, possuem seu escopo limitado a esse estado. Assim, o conjunto de propriedades que o estado *PHYUP* deve satisfazer é formado pelas propriedades locais do seu estado, juntamente com as propriedades globais.

Resumidamente, o trecho da máquina de estados hierárquica da figura 2.10, juntamente com as propriedades temporais mostradas na figura 2.14, podem ser visualizadas na figura 2.15. Além das propriedades mostradas na figura 2.14, que são locais ao estado *PHYUP*, foram adicionadas outras propriedades pertencentes ao estado global (*globalstate*).

2.4 Provadores automáticos de teoremas

Satisfatibilidade é um dos problemas mais fundamentais em informática teórica e consiste em determinar se uma fórmula que expressa uma restrição tem solução BJORNER; MOURA

Figura 2.15: Visualização gráfica dos trecho da HFDM-D mostrados na figura 2.10 e 2.14



(2009). O mais conhecido problema de satisfatibilidade de restrição é o Problema da Satisfatibilidade Booleana (chamado também de Problema da Satisfatibilidade Proposicional e abreviado para Satisfatibilidade ou SAT), no qual o objetivo é decidir se uma fórmula, formada por variáveis booleanas e conectivos lógicos, pode ter seu valor *verdadeiro*, escolhendo os valores *verdadeiro* ou *falso* para suas variáveis. Por exemplo a fórmula x $\wedge \neg$y formada pelas variáveis booleanas x e y é satisfatível, pois será verdadeira quando o valor de x for *verdadeiro* e o valor de y for *falso*, conforme pode ser verificado na tabela verdade 2.1.

Tabela 2.1: Exemplo de uma fórmula booleana satisfatível

x	y	x $\wedge \neg$y
V	V	F
V	F	V
F	V	F
F	F	F

Alguns problemas, porém, requerem ou são mais naturalmente descritos em uma lógica mais expressiva, como a lógica de primeira ordem. Uma fórmula, na lógica de primeira ordem, é formada usando conectivos lógicos, variáveis, quantificadores, funções e símbolos de predicados. Uma solução, também conhecida como modelo, é uma interpretação para as variáveis, funções e símbolos de predicados que tornam a fórmula verdadeira.

Para auxiliar na busca dessa solução, tem-se o problema das Teorias dos Módulos da Satisfatibilidade (STM, sigla em inglês para Satisfiability Modulo Theories) que verifica se uma determinada fórmula lógica F é satisfatível usando a combinação de diferentes teorias de fundamentação (background). Por exemplo, a teoria da aritmética restringe-se à interpretação de símbolos, como +, $\leq$,0 e 1.

Teoria:

Uma teoria é essencialmente um conjunto de sentenças. Mais formalmente, um $\sum$-teoria é uma coleção de sentenças sobre uma assinatura $\sum$. Uma *valoração-verdade M* para uma fórmula φ mapeia as variáveis proposicionais de φ para um valor no conjunto *true*,*false*. Dessa forma, dizemos que uma *valoração-verdade M* satisfaz φ (M $\models \varphi$), se *M* torna φ verdadeira dado os valores atribuídos a cada variável. Por exemplo, seja a fórmula φ igual a $p \vee (\neg q \wedge r)$, então a *valoração-verdade* M = (p $\rightarrow$*false*, q $\rightarrow$*false*, r $\rightarrow$*true*) satisfaz φ MOURA; BJORNER (2009).

Dada uma teoria T e uma fórmula φ, dizemos que φ é satisfatível módulo T se T $\cup \{\varphi\}$ é satisfatível. Usamos M $\models_T \varphi$ para denotar que M $\models \{\varphi\} \cup$ T. Por exemplo, seja $\sum$ a assinatura contendo os símbolos 0, 1, +, - e <, e $\mathbb{Z}$ o conjunto dos números inteiros, então, a teoria da aritmética linear é o conjunto de sentenças de primeira ordem, que são verdadeiras em $\mathbb{Z}$.

Um solucionador de problemas (*solver*, em inglês) STM é uma ferramenta para decidir a satisfatibilidade de fórmulas no contexto de alguma teoria. Solucionadores STM podem ser usados em diversas aplicações, tais como verificação estática estendida , abstração de predicados e geração de casos de testes BJORNER; MOURA (2008). Eles podem ser compostos por diversas

teorias, algumas que merecem destaque são:

- Aritmética linear, também conhecida como aritmética aditiva: é a teoria na qual as únicas funções aritméticas são a soma (+) e a subtração (-). As funções podem ser aplicadas a qualquer constante numérica ou variável.
- Aritmética da diferença: é uma fragmentação da aritmética linear, onde predicados são restritos a serem da forma x - y $\leq$ c, para x e y variáveis e c uma constante numérica.
- Vetores de bits: a aritmética de máquinas não é a mesma do que os inteiros da matemática. Na aritmética de máquinas, os inteiros se encaixam em registradores de tamanho fixo. Um domínio mais adequado para a aritmética de máquina é representar cada número como uma sequência de tamanho fixo de bits. Em uma CPU de 64 bits, por exemplo, um número inteiro é representado por um vetor de bits de 64 bits.

Solucionador Z3:
Z3 é um solucionador da Microsoft Research e está disponível gratuitamente para fins de pesquisa acadêmica. Ele integra uma série de teorias em uma combinação expressiva e eficiente. Uma breve descrição do sistema Z3 está disponível em BJORNER; MOURA (2008).

Por não possuir uma interface muito amigável e para facilitar o uso em diversas aplicações, normalmente Z3 é integrado a outras ferramentas através de sua API ou através da SMT-lib. Nesse trabalho, usamos a API na linguagem de programação de alto nível Python.

2.5 Conclusão

Neste capítulo foi apresentado o contexto no qual o trabalho está inserido assim como os conceitos que precisam ficar claros para o posterior entendimento da abordagem proposta.

As definições de LTL e BA devem estar claras o suficiente para o entendimento no capítulo 4 da transformação de LTL em um autômato de Büchi, passo crucial para entender como a checagem de inconsitências na HFSM-D é realizada.

3

Trabalhos Relacionados

Neste capítulo serão apresentados os trabalhos relacionados. Os trabalhos descritos apresentam técnicas para encontrar contradições nas propriedades temporais de um modelo, como também detectar a presença de não-determinismos.

3.1 Cleaveland 2002 - Automated Validation of Software Models

O trabalho apresentado em SIMS; BUTTS; RANVILLE (2002) descreve um projeto de engenheiros da Ford e da Reactive Systems, Inc. (RSI) para investigar como ferramentas de verificação automática podem ser empacotadas mais eficazmente para serem rapidamente integradas em um processo de desenvolvimento de software industrial como os utilizados pela Ford.

Ford já possui em vigor um avançado framework de desenvolvimento de software baseado em modelo que usa o Matlab, Simulink e Stateflow, componentes de um ambiente de modelagem da engenharia construído e comercializado pela MathWorks, Inc. Para o trabalho apresentado, foi utilizado um modelo que especifica o comportamento de um pedaço do software do controlador de um *powertrain* (o powertrain de um carro é composto por muitos componentes, incluindo o motor, transmissão, eixo do motor e qualquer um dos trabalhos internos de um motor).

Simulink é uma ferramenta de diagramação gráfica por blocos e bibliotecas customizáveis de blocos. Ela permite a especificação de restrições matemáticas nos valores de entrada e saída de variáveis. Além disso, fornece um meio para a especificação de decomposições hierárquicas de um sistema em subsistemas de forma mais simples. Stateflow, uma versão da notação Statecharts, suporta a especificação do comportamento de um componente do sistema em termos dos estados que o componente pode entrar e as transições indicam como os componentes evolui de um estado para outro.

A Ford implementou um avançado processo de desenvolvimento de software baseado em modelo em que os benefícios da ferramenta vão desde o estágio de levantamento de requisitos até o teste do sistema. O processo está centrado no desenvolvimento de modelos executáveis durante

as fases iniciais. Experiências preliminares mostraram que um esforço extra necessário para a modelagem é recuperado de várias maneiras, como por exemplo a possibilidade de validação desde o início do desenvolvimento, permitindo, dessa forma, que problemas sejam detectados antes, quando são menos caros para consertar.

A ferramenta utilizada para realizar a análise do modelo foi Salsa BHARADWAJ; SIMS (2000), um verificador de invariantes para os modelos especificados em SAL (SCR - Software Cost Reduction- Abstract Language). Uma invariante é uma afirmação lógica que é sempre mantida como verdadeira durante um certo período de execução. Para verificar se uma fórmula é uma invariante de um modelo SAL, Salsa realiza uma prova de indução que utiliza procedimentos de decisão fortemente integrados, atualmente uma combinação de algoritmos BDD (Binary Decision Diagram) e um solucionador de restrições para a aritmética linear de inteiros. Salsa possui atributos tanto de um verificador de modelos como de um provador de teoremas. A principal desvantagem de Salsa é a sua incompletude, uma verificação que falhou não necessariamente implica que a fórmula não é uma invariante porque o par de estados rertornado pode simplesmente não ser acessível.

Os tipos de checagem realizadas no modelo utilizando Salsa foram:

- Não-determinismo;
- Não consideração de todos os casos possíveis;
- Violações de exclusão mútua; e
- Código morto e redundante

Como uma das propostas dessa dissertação é a checagem de um modelo em busca de não-determinismos, explica-se com mais detalhes como SIMS; BUTTS; RANVILLE (2002) realiza essa checagem.

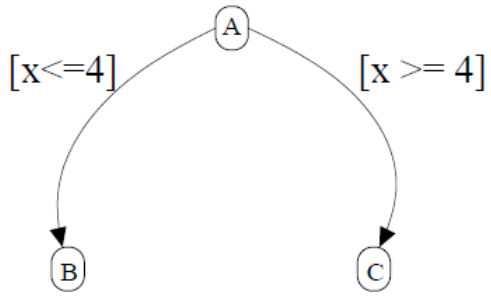
3.1.1 Não-determinismo

Não-determinismo é uma construção de modelagem útil muitas vezes para a especificação de incertezas em ambientes onde software embarcados operam. Por isso, várias linguagens de modelagem, incluindo SAL, possuem construções para a representação de não-determinismo. Porém, na maioria dos casos, softwares em sistemas embarcados devem se comportar de maneira determinística, ou seja, para uma determinada entrada deve haver apenas uma resposta possível. Assim, verificar o modelo de um software para não-determinismo é bastante útil.

Considerando, por exemplo, o fragmento de um diagrama que contém três estados e duas transições, mostrado na figura 3.1, tem-se que as condições mostradas entre colchetes nas transições são as condições de guarda, que indicam quando a transição rotulada pode disparar. Se o diagrama está no estado A, então ele pode ir pro estado B se $x \leq 4$ ou para o estado C se $x \geq 4$. É possível verificar que no caso de $x = 4$ o sucessor de A pode ser B ou C. Pode-se,

contudo, procurar os casos de não-determinsmo usando verificadores de invariantes. Nesse caso do exemplo mostrado, uma invariante proposta pode ser $\neg(x \leq 4) \vee \neg(x \geq 4)$. Claramente, uma vez que ambas as disjunções são falsas quando $x = 4$, a fórmula não é uma invariante do diagrama e portanto, não-determinismos é possível.

Figura 3.1: Fragmento de um diagrama Stateflow com a presença de não-determinismo



Fonte: SIMS; BUTTS; RANVILLE (2002)

A checagem de não-determinismos apresentada nesse trabalho, apesar de eficaz, no sentido de encontrar os casos que a inconsistência acontece, possui a tradução do modelo de MathWorks para SAL de forma manual o que revela ser uma grande desvantagem da abordagem proposta.

3.2 Toyn 2007 - Formal Validation of Hierarchical State Machines against Expectations

Em projetos baseados em modelos, modelos devem ser validados contra as intenções e o código deverá ser verificado contra o modelo. O trabalho apresentado em TOYN; GALLOWAY (2007) preocupa-se com a validação de modelos que são construídos como Máquinas de Estados Finitas Hierárquicas. Ele explica algumas análises que podem ser realizadas em uma Máquina de Estados Finita Hierárquica para validá-la para que o seu comportamento seja conforme o esperado. Cada estado da máquina pode ser modelado com as expectativas do projetista, então, a análise determina onde essas expectativas são consistentes e onde a máquina de estados está em conformidade com essas expectativas.

Uma expectativa é uma condição sobre os valores de entrada, variáveis locais e constantes. Cada estado pode ter várias expectativas anotadas nele. Essas expectativas podem servir diferentes papéis, como por exemplo, uma expectativa pode dizer quais os valores da entrada que são esperados para o próximo estado, enquanto outra pode dizer quais desses valores de entrada faria com que o estado permanecesse ativo.

Quando a ferramenta que implementa a análise acha uma inconsistência ou uma não-conformidade, um contra-exemplo pode ser relatado. Esse contra-exemplo serve para orientar o projetista para corrigir a máquina de estados ou para consertar a expressão das expectativas.

A máquina de estados hierárquica para a qual a análise é definida nesse artigo, compreende estados básicos, agrupados hierarquicamente por estados ou-exclusivo. Quando um estado possui uma decomposição exclusiva, somente um dos subestados podem ser ativados por vez. Os estados são unidos por transições e cada transição possui suas condições de disparo sobre entradas, variáveis locais e constantes locais, as quais podem ser do tipo inteiro ou booleano. As condições de disparo das transições podem usar notações para disjunção ($|$), conjunção (&) e negação ($\sim$). A semântica para essas máquinas de estados é de transição seguida de computação: as entradas, variáveis locais e constantes podem desencadear uma transição, e nesse caso, a configuração corrente do estado ativo fica sendo determinada pelo estado de estado da transição. Em seguida, novos valores são calculados para as saídas e variáveis locais de acordo com a última configuração dos estados ativos.

A máquina de estados hierárquica pode apresentar erros como: condições inapropriadas das condições das transições de disparo; falta de transições; e a falta ou a contradição de expectativas. Dessa forma, as análises podem ser realizadas na máquina de estados para verificar o(a):

- Estabelecimento das premissas iniciais;
- Estabelecimento das próximas premissas;
- Estabelecimento das últimas premssissas;
- Preservação das últimas premissas;
- Transições de saída disjuntas; e
- Transições de saída completas.

As análisis são realizadas automaticamente por extensões da ferramenta Stateflow da MathWorks. O usuário escolhe quando iniciar a análise que são realizadas por meio de provadores de teoremas autiméticos. Quando a ferramenta encontra um problema, o usuário é notificado e usualmente um contra-exemplo é apresentado.

De todas essas análises, a que merece destaque para a proposta apresentada nessa dissertação é a análise das transições de saída disjuntas. Statecharts permitem que variáveis de entrada disparem mais de uma transição de saída saindo do mesmo estado. Stateflow resolve este não determinismo primeiramente priorizando transições que causam a saída de um estado e por último pegando a primeira transição encontrada em uma busca no sentido horário em torno do perímetro do estado iniciando-se no canto superior esquerdo. Ou seja, pode existir não-determinismo, que é resolvido prioprizando uma transição por vez e essa priorização depende em

grande escala do layout da máquina de estados. Isso pode ser evitado exigindo que as condições das transições se saída sejam disjuntas. Para essa análise, primeiramente a máquina de estados é transformada em uma máquina equivalente na qual todas as transições são de e para estados básicos (estados que não são compostos por outros estados). Depois, as seguintes análises são realizadas:

- Para qualquer estado básico S na máquina de estados equivalente e para quaisquer transições distintas T e U, ambas existentes em S temos que:

  $'lasts(S)$ & $next(S)$ & $local(S) \Rightarrow \sim (trigger(T)$ & $trigger(U))$

  Onde $'lasts(S)$ refere-se aos útimos valores da entrada. $Next(S)$ especifica quais são os próximos valores que as entradas devem possuir enquanto estiver no estado. $Local(S)$ é a conjunção das definições locais (em que cada definição local é uma igualdade entre o próximo valor de uma variável local e uma expressão em termos dos seu seus últimos valores, por exemplo, a igualdade $x = x+1$ torna-se a igualdade $x ==' x+1$). E finalmente, para cada transição T, sua condição de disparo é $trigger(T)$.

- Para qualquer estado básico S na máquina de estados equivalente e para qualquer transições T existente em S temos que:

  $'lasts(S)$ & $next(S)$ & $local(S) \Rightarrow \sim (trigger(T)$ & $spc(S))$

  Onde $spc(S)$ especifica quais são os próximos valores das entradas que fazem com que não haja uma saída do estado.

Quando uma análise é iniciada, a máquina de estados hierárquica e suas expectativas são transformadas na notação formal Z Z STANDARDS PANEL (2000), conjecturas Z para todas as análises são geradas, e a prova para a análise solicitada é realizada pelo provador de teorema CADiZ. Quando uma outra análise é iniciada no mesmo modelo, a máquina na notação Z é reusada. Porém, na análise de grandes modelos, algumas análises são realizadas mais de uma vez.

3.3 Felty and Namjoshi 2000 - Feature Specification and Automated Conflict Detection

Grandes sistemas de software, especialmente no campo das telecomunicações, são muitas vezes especificados como uma coleção de funcionalidades. Esse artigo apresenta uma linguagem de especificação formal para descrever funcionalidades, e um método para detectar automaticamente conflitos (interações indesejáveis) entre essas funcionalidades no estágio da especificação do sistema.

A detecção de conflitos nessa fase inicial pode ajudar a evitar correções de problemas dispendiosos e prolongados encontrados na fase de implementação. As funcionalidades são

especificadas usando a lógica temporal. Duas funcionalidades são conflitantes se, essencialmente, suas especificações são mutuamente inconsistentes sob os axiomas definidos sobre o comportamento do sistema.

O artigo FELTY; NAMJOSHI (2013) mostra como essa verificação de inconsistência pode ser verificada automaticamente com ferramentas de verificação de modelos (Model Cheking) existentes. Foi desenvolvida uma ferramenta de detecção de conflitos, FIX (Feature Interaction eXtractor), que usa o verificador de modelos COSPAN para a checagem de inconsistências. Os experimentos foram realizados em uma coleção de especificações de funcionalidades da áres de telecomunicações. Por exemplo, uma típica especificação informal para o encaminhamento de chamadas é "Se uma entidade x tem o encaminhamento de chamadas habilitado e as chamadas para x devem ser encaminhadas para z então, sempre que x estiver ocupado, qualquer chamada de y para x, eventualmente, é encaminhada para z". Essa descrição informal pode ser expressa precisamente na linguagem de especificação proposta. Essa linguagem pode ser vista como uma versão mais incrementada da lógica temporal ou ω-autômato (autômato que reconhece uma linguagem ω-regular). Especificar funcionalidades como fórmulas temporais tem como vantagem a abstração da implementação em uma máquina de estados específica.

O caminho natural para definir um conflito de funcionalidade é que a especificação das funcionalidades representa propriedades multuamente inconsistentes. Ou seja, não existe nenhum programa que implemente ambas as funcionalidades. Essa é uma questão sobre se a conjunção de duas especificações de funcionalidades é realizável.

3.3.1 Especificação da funcionalidade

A especificação de funcionalidades inicia-se com a descrição informal, principalmente na forma de texto em inglês. Naturalmente, o processo de passar da especificação informal para a formal não pode ser formalizado. Dessa forma, deve-se tomar muito cuidado para expressar corretamente o conteúdo da descrição informal.

Uma funcionalidade, como chamada em espera ou encaminhamento de chamada, normalmente especifica o comportamento ao longo do tempo de uma ou mais entidades em termo do seu estado atual e um conjunto de eventos de entrada. A especificação informal dada anteriormente para o encaminhamento de chamadas é um exemplo: "Se uma entidade x tem o encaminhamento de chamadas habilitado e as chamadas para x devem se encaminhadas para z então, sempre que x estiver ocupado, qualquer chamada de y para x, eventualmente, é encaminhada para z". Nesta especificação, pode-se distinguir vários predicados que descrevem o estado da entidade x: *encaminhamento_chamada_habilitado(x)*, *encaminhamento_de_para(x,y)*, *chamadaDesviada_de_para(y,x,z)*, *ocupado(x)* e o predicado *chamada_recebida_de_para(y,x)* que descreve a ocorrência de um evento. O restante da sentença usa operadores booleanos e temporais (por exemplo, "E", "sempre", "eventualmente"). Dessa forma, acredita-se que a melhor forma de especificar uma funcionalidade é por meio de uma coleção de fórmulas temporais (ou autômatos),

definidos por um conjunto de predicados que denotam estados ou eventos do sistema.

Cada funcionalidade é especificada separadamente como uma coleção de propriedades temporais. As propriedades são definidas em termos de predicados que indicam relações entre entidades do sistema. A forma geral de especificar uma propriedade é mostrada abaixo:

property <Name>
{
event: e0 persists: p0
event: e1 persists: p1
...
evento: eN
- - - - - - - - - - - - - - -
persists: p until: r discharge: d
}

Os símbolos e0, p0, e1, p1,..., eN, p, r, d são expressões booleanas constituídas a partir dos predicados básicos. Variáveis como x, y que aparecem nos predicados da especificação de uma propriedade possuem escopo local, e são implicitamente universalmente quantificadas, isto é a propriedade temporal deve ser verdadeira para todo o valor de x, y em um sistema particular. As condiçẽes de evento (event) e persistência (persists) acima da linha tracejada indicam a pré-condição da propriedade; o trio persists-until-discharge indica a pós-condição da propriedade. Informalmente, a propriedade afirma que "sempre que o padrão de pré-condição for verdadeiro, ele é seguido por um padrão de pós-condição".

A pré-condição tem a seguinte leitura informal: "o evento e0 acontece, seguido por um período onde (p0 $\wedge\neg$ e1) é verdadeiro, então o evento e1 acontece, seguido por um período onde (p1 $\wedge\neg$ e2) é verdadeiro, etc., até que o evento eN acontece". Por padrão, uma pré-condição vazia indica que ela é verdadeira. A pós-condição deve acontecer em todos os pontos da computação onde a propriedade é habilitada. Na notação LTL, a pós-condição é representada por: $p \bigcup (r \vee d)$. Embora a condição de descarga (*discharge*) possa parecer tecnicamente desnecessária, ela faz uma distinção importante para o especificador. A condição *until* é pensada como se estivesse especificando um resultado desejado, enquanto a condição *discharge* é pensada como se estivesse especificando uma condição de exceção que faça com que a propriedade seja trivialmente satisfeita.

A maneira mais fácil de definir completamente uma propriedade em LTL associada com a forma geral é considerando a sua negação: a propriedade é falsa em uma sequência infinita se e somente se existe um ponto onde a pré-condição acontece, mas não é seguida pelo padrão da pós-condição. Para ilustrar a tradução, considere a propriedade abaixo:

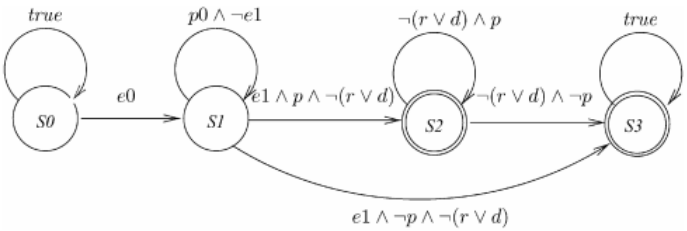
property Simple
{
event: e0 persists: p0 event: e1
- - - - - - - - - - - - - - -

persists: p until: r discharge: d

}

A propriedade LTL $\neg F(e0 \wedge X((p0 \wedge \neg e1) \bigcup (e1 \wedge \neg(p \bigcup (r \vee d)))))$ é equivalente a essa especificação. F e X são respectivamente os operadores da lógica temporal $\Diamond$ e $\bigcirc$ vistos na seção 2.2 do capítulo 2. A negação da propriedade *Simple* pode ser expressa por um BA, como mostra a figura 3.2.

Figura 3.2: Autômato da negação da propriedade *Simple*



Fonte: FELTY; NAMJOSHI (2013)

Na figura 3.2, os estados são representados por círculos, a relação de transição é definida pelas condições nos arcos entre os círculos e os estados de aceitação são representados por círculos concêntricos. O estado marcado com S0 é o estado inicial. O autômato no estado S0 escolhe (não-deterministicamente) algum ponto da computação, verifica que a pré-condição acontece a partir desse ponto (estados S1 e S2), e que a pós-condição falha posteriormente (isto é, o autômato fica preso nos estados S2 e S3).

3.3.2 Detecção de funcionalidades conflitantes

As funcionalidades A e B são conflitantes se e somente se não existe um sistema, ou programa, no qual todas os caminhos computacionais satisfazem ambas as especificações de A e B. Assim, funcionalidades conflitantes é essencialmente uma questão de capacidade de realização. Percebe-se, portanto, que o interesse é justamente saber se esse programa existe. O problema de sintetizar esse programa é um clássico problema que tem potenciais aplicações práticas.

Qualquer programa que satsisfaça A e B será um programa reativamente aberto que constantemente interage com o seu ambiente. Por exemplo, um programa que satisfaça a especificação de encaminhamento de chamadas terá de responder a eventos de chamadas recebidas e produzir eventos de chamadas de saída. Infelizmente, a questão da realizabilidade para programas reativamente abertos tem uma complexidade muito elevada, e as soluções conhecidas são baseadas em mostrar a satisfabilidade da ramificação de uma fórmula temporal obtida a partir das fórmulas lineares no tempo descrevendo A e B. Abaixo tem-se a definição formal apresentada nesse artigo:

Definição. ***Funcionalidades conflitantes***

Funcionalidades A e B são conflitantes sse A e B podem ser habilitadas juntas infinitamente sobre os axiomas do sistema, e para toda computação onde:

1. *Os axiomas do sistemas são mantidos, e*
2. *A e B são habilitadas infinitas vezes em conjunto, e*
3. *A condição de descarga (discharge) não acontece enquanto a funcionalidade está pendente,*

alguma propriedade da funcionalidade não se mantém.

Cada teste de conflito é realizado em uma específica instância das funcionalidades. A forma parametrizada da espeficicação das funcionalidades torna mais fácil a instanciação de diferentes configurações.

Sendo $L(x)$ a linguagem aceita pela propriedade LTL x, temos que em geral, duas propriedades LTL f e g são inconsistentes se e somente se $L(f) \cap L(g) = \emptyset$, que é verdadeira se e somente se $L(f) \subseteq \overline{L(g)}$. Isto é exatamente a questão do verificador de modelos com f como um programa e $\neg g$ como a propriedade. Assim, verificadores de modelo podem ser usados para detectar funcionalidades conflitantes.

Sejam A e B duas funcionalidades, A_x denota os axiomas do sistema e C_{AB} as restrições dadas pelas condições (2) e (3) da definição de funcionalidades conflitantes acima apresentada. A verificação da inconsisência pode ser escrita como: $L(A) \cap L(B) \cap L(A_x) \cap L(C_{AB}) = \emptyset$, que é equivalente a $L(A_x) \cap L(C_{AB}) \subseteq \overline{L(A)} \cup \overline{L(B)}$. Essa é a forma usada na implementação proposta desse artigo.

A ferramenta desenvolvida, FIX, usa o verificador de modelos COSPAN H.; Z.; KURSHAN (1996) para a verificação de conflito. Em COSPAN, ambas as propriedades e restrições são representadas por um ω-autômato. FIX traduz as restrições A_x e as especificações das funcionalidades A e B em autômatos COSPAN que aceitam as linguagens especificadas. Cada funcionalidade é traduzida para um autômato parametrizado (parametrizado pelas variáveis que aparecem na propriedade), que é instanciado conforme necessidade em cada teste em particular. Uma vez que os autômatos representando as condições (2) e (3), da definição apresentada na seção 3.3.2, são independentes da fucionalidade em particular, eles são obtidos a partir de uma biblioteca e instanciado em cada uso com a condição de habilitação das funcionalidades em particular para obter o autômato C_{AB}.

O verificador de modelos declara uma falha se o conjunto acima apresentado ($L(A_x) \cap L(C_{AB}) \subseteq \overline{L(A)} \cup \overline{L(B)}$) for falso, isto é, se as propriedades não entram em conflito. Uma vez que o verificador de modelos declara uma falha, ele produz uma prova computacional na qual os axiomas e as funcionalidades podem ser verdadeiras ao mesmo tempo. Inspeções nessa prova computacional podem revelar restrições que precisam ser incluídas nos axiomas do sistema. Mesmo que não seja o caso, um relatório sobre a ausência de conflitos pode ser, em geral,

considerado inconclusivo, uma vez que a checagem é realizada em uma configuração particular do sistema.

Por outro lado, um resultado de presença de conflito é conclusivo, porém, como o verificador de modelos declara sucesso, nenhuma prova é produzida para o conflito. Para produzir uma prova, executa-se outra verificação: $L(A_x) \cap L(C_{AB}) \cup L(A) \subseteq \overline{L(B)}$. Como há um conflito essa checagem deve falhar, então o verificador de modelos produz uma computação que satisfaz A_x, C_{AB} e A, mas não satisfaz B. Essa computação descreve um cenário no qual os axiomas do sistema são mantidos, as duas funcionalidades são habilitadas juntas infinitamente, a funcionalidade A é mantida, mas B não.

3.3.3 FIX: A ferramenta de detecção de conflitos

A ferramenta FIX é usada tanto para especificar as propriedades das funcionalidades quanto para detectar conflitos entre essas funcionalidades. FIX destina-se a ser usada no estágio de projeto e especificação do desenvolvimento de novas funcionalidades.

O primeiro passo para usar FIX é fornecer um conjunto de propriedades que especifica o comportamento desejado de uma nova funcionalidade. Não há necessidade de saber as linguagens da lógica linear temporal ou do autômato de Büchi nas quais as propriedades irão ser traduzidas. Ao desenvolver uma nova especificação, o usuário tem a liberdade de introduzir novos predicados, conforme for necessário. A introdução de novos predicados usualmente requer que novos axiomas do sistema sejam adicionados ou atualizados.

A verificação de conflito é a operação central da ferramenta FIX. Existem dois tipos de verificação: a checagem de inconsistência e a checagem que produz uma prova do conflito, uma vez que a inconsistência foi detectada. Para ambos os tipos de verificação, FIX espera duas propriedades, A e B, como entrada. Os axiomas do sistema A_x são fixados e o autômato auxiliar, C_{AB}, é criado automaticamente a partir de A e B. O segundo passo para usar FIX é usar o primeiro tipo de checagem como uma ajuda de depuração. Em particular, em cada propriedade de uma nova funcionalidade pode ser checada a existência de conflitos diretamente com os axiomas do sistema. Para executar essa verificação de uma única propriedade A com os axiomas do sistema, simplemente instancia-se B com a propriedade "sempre verdadeira".

Uma vez que a checagem de conflito é conclusiva, dado que a fase de especificações inicial foi concluída, o usuário pode ter a certeza de que não existem conflitos entre as propriedades das funcionalidades e os axiomas do sistema. Por outro lado, dado que um resultado de "nenhum conflito"é inconclusivo, não há garantia de que todos os potenciais conflitos entre as funcionalidades serão encontrados.

Uma vez que o usuário obteve garantias suficientes de que a especificação e os axiomas do sistema estão corretos, propriedades das novas funcionalidades podem ser checadas juntamente com todas as outras funcionalidades automaticamente. Nesta fase, apenas os conflitos são importamtes e as provas dos conflitos são úteis para a compreensão da determinação de como

corrigi-los.

3.3.4 Estudo de caso

A ferramenta FIX foi aplicada em uma coleção de especificações de funcionalidades derivada dos padrões Telcordia TELCORDIA-BELLCORE (1996). Foram relatados os resultados de dez essas funcionalidades, cada uma verificada com as outras nove.

A figura 3.3 representa a tabela extraída do artigo e descreve as dez funcionalidades, assim como a quantidade de propriedades de cada uma.

Figura 3.3: Funcionalidades, número de propriedades usadas na especificação e descrição

ACR	Anonymous Call Rejection	6	Allows subscriber to reject calls from parties who have a privacy feature that prevents the delivery of their calling number to the called party. When active, the call is routed to a denial announcement and terminated.
CFBL	Call Forwarding Busy Line	3	A telephone-company-activated feature that forwards incoming calls to a subscriber to another line when the subscriber is busy.
CFDA	Call Forwarding Don't Answer	4	Incoming calls to the subscriber are forwarded when the subscriber doesn't answer after a specified time interval.
CFMB	Call Forwarding Make Busy	1	Allows subscriber to press a key to put phone into a busy state so that all calls will be forwarded.
CFV	Call Forwarding Variable	7	Allows subscriber to specify a number to which all calls will be forwarded.
CW	Call Waiting	16	Informs a busy subscriber that another call is waiting by playing a tone. The subscriber may flash, placing the original call on hold and answer the new call, or may go on hook, in which case the subscriber is rung and connected to the new call upon answer.
DOS	Denied Originating Service	2	Provides the capability to deny a subscriber from making calls.
DTS	Denied Terminating Service	2	Provides the capability to deny terminating calls to a subscriber.
PKUP	Call Pickup	2	Allows one station to answer a call directed to another station within a business group.
RDA	Residential Distinctive Alerting	2	Allows the subscriber to designate special telephone numbers that may be identified using distinctive alerting treatment.

Fonte: FELTY; NAMJOSHI (2013)

As funcionalidades são consideradas em pares, e cada propriedade de uma funcionalidade é verificada em relação a cada propriedade da outra funcionalidade. As verificações são realizadas utilizando um banco de dados com cerca de 50 axiomas expressos como restrições.

A figura 3.4 mostra os resultados das verificações das dez funcionalidades. Os números indicam a quantidade de pares de propriedades que resultaram em conflito quando uma funcionalidade foi checada em relação a outra. Agumas entradas estão em branco para evitar duplicação. Os resultados relatados foram realizados usando as configurações padrão da ferramenta FIX.

3.3.5 Trabalhos relacionados e conclusão

Várias abordagens têm sido propostas para o problema de detecção de conflito. Existem duas principais categorias baseadas no formalismo da especificação: método baseado em máqui-

Figura 3.4: Número de pares de propriedades conflitantes de cada par de funcionalidades

	CFBL	CFDA	CFMB	CFV	CW	DOS	DTS	PKUP	RDA
ACR	8	5	4	3	8	2	4	4	0
CFBL	—	0	2	2	4	1	0	2	0
CFDA	—	—	2	4	0	0	2	0	0
CFMB	—	—	—	3	0	1	1	0	0
CFV	—	—	—	—	2	1	2	1	0
CW	—	—	—	—	—	0	2	1	0
DOS	—	—	—	—	—	—	0	3	0
DTS	—	—	—	—	—	—	—	1	0
PKUP	—	—	—	—	—	—	—	—	0

Fonte: FELTY; NAMJOSHI (2013)

nas de estado e o método baseado na lógica temporal. O artigo discutido nessa seção se enquadra na categoria da lógica temporal.

Em vários métodos de especificação, cada funcionalidade é especificada por uma máquina de estados. Interações são detectadas testando a composição das máquinas, quer para a acessibilidade de estados de exceção, ou dos estados onde as funcionalidades postulam ações conflitantes em uma nova entrada, ou contra propriedades temporais que especificam o comportamento da funcionalidade.

Nesse trabalho, foi descrito um método para detectar conflitos de funcionalidades onde as funcionalidades são especificadas como um conjunto de fórmulas na linguagem temporal ou ω-autômato, e as interações são descobertas encontrando pares de fórmulas da especificação que são contraditórias com relação aos axiomas sobre o comportamento do sistema.
Foi mostrado que verificadores de modelos existentes podem ser usados para executar este teste. As principais vantagens dessa abordagem são:

- A linguagem de especificação simplifica a manutenção das especificações;
- O método evita qualquer compromisso com uma implementação em particular, o que significa que a detecção de conflito pode ser aplicada a todas as implementações;
- Pode ser implementado de forma eficaz para realizar a detecção de conflitos totalmente automatizado, usando verificadores de modelos existentes.

O método foi implementado e aplicado na análise de especificações formais derivadas do padrão Telcordia. A experiência mostrou que este processo de detecção de conflitos é razoavelmente eficiente e bastante preciso. Para o conjunto de funcionalidades que o método foi aplicado, foi possível detectar a maioria das interações. Para o conjunto de funcionalidades testadas, a ferramenta FIX foi capaz de detectar estas interações em algumas horas de tempo de processamento.

3.4 Conclusão

Fazendo uma análise aprofundada dos trabalhos acima apresentados, verfica-se que o trabalho apresentado em TOYN; GALLOWAY (2007), seção 3.1 apresenta uma ferramenta para a

da presença não-determinismos em máquinas de estados hierárquicas, entre outras inconsistências. Porém, os tipos de dados definidos para as variáveis e constantes de entrada são inteiro e booleano apenas. Isso é uma grande limitação pois na especificação TDevC existem os tipos de dados *register* e *pattern* que são variáveis de entrada e precisam de um tipo de dado mais complexo para sua representação, como vetores de bits. Já o trabalho apresentado em SIMS; BUTTS; RANVILLE (2002), seção 3.2, além de não possuir uma representação para um tipo mais complexo das variáveis de entrada, a transformação do modelo Stateflow para a linguagem SAL é realizada de forma manual. Além disso, as duas abordagens apresentadas necessitam de um modelo prévio em diagramas Stateflow.

O trabalho apresentado em FELTY; NAMJOSHI (2013), seção 3.3, apresenta uma abordagem para encontrar propriedades temporais conflitantes a partir de verificadores de modelos (model checker). O artigo mostra que duas propriedades temporais f e g são inconsistentes se $L(f) \cap L(g) = \emptyset$. Testes foram realizados no verificador de modelos SPIN HOLZMANN (1997) a fim de validar a proposta, porém, o autômato gerado da conjunção de duas propriedades temporais contraditórias, apesar de ser vazio (não possuir nenhum estado de aceitação) não apresenta um autômato simplificado, dificultando a interpretação. Por exemplo, sendo f = □(a && b) (sempre *a* e *b* são verdadeiros) e g = ◇!b (eventualmente no futuro *b* é falso), o resultado do autômato gerado pela conjunção dessas duas propriedades contraditórias apesar de possuir uma linguagem vazia (ausência do estado de aceitação do autômato), não é apresentado de uma forma simplificada, como mostra a figura 3.5. Uma vez que a forma simplificada para esse caso seria a indicação de um autômato vazio simplesmente retornando a palavra *empty language*.

```
1 T0_init :
2   if
3   :: ((a) && (b)) -> goto T0_init
4   fi;
```

Figura 3.5: Autômato resultante da conjunção de duas propriedades temporais contraditórias na linguagem PROMELA

4

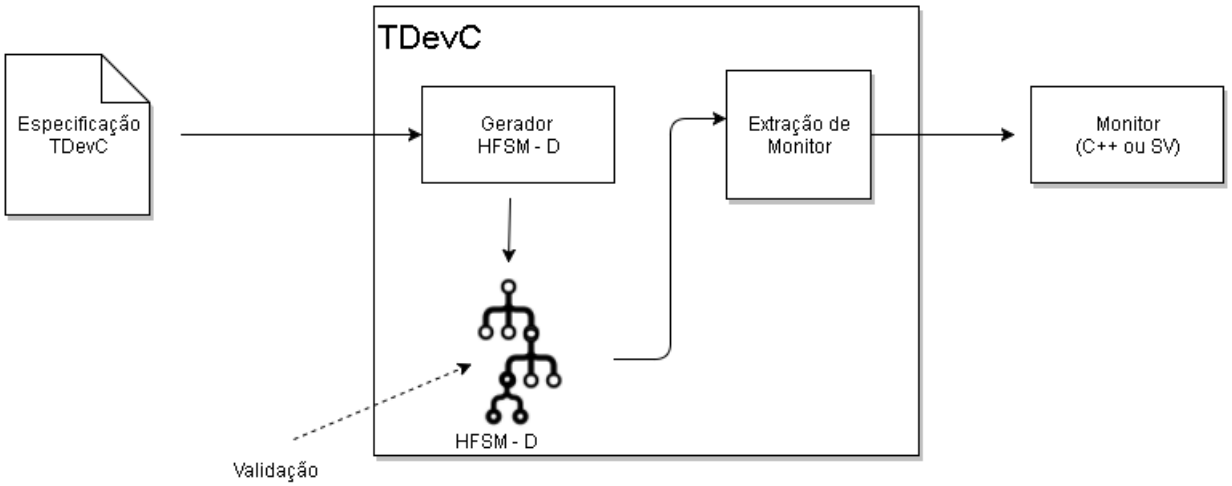
Estratégia proposta para a checagem de não-determinismo e contradição

Neste capítulo é apresentada a abordagem proposta, que tem como objetivo principal detectar a presença de não-determinismos e de propriedades temporais contraditórias em uma especificação TDevC de um dispositivo.

4.1 Visão geral

O objetivo do presente trabalho é a validação de uma especificação em TDevC, que dará suporte à técnica proposta em MACIEIRA; BARROS; ASCENDINA (2014). Essa validação busca auxiliar o desenvolvimento de drivers de dispositivos robustos e confiáveis através da identificação de violações de propriedades dos protocolos de comunicação de alto nível.

Figura 4.1: Visão geral



A figura 4.1 apresenta uma visão geral da técnica proposta em MACIEIRA; BARROS; ASCENDINA (2014) com a inserção da etapa de validação da HFSM-D que será proposta mais adiante. A primeira etapa do fluxo da técnica proposta em MACIEIRA; BARROS; ASCENDINA

(2014) consiste na especificação em TDevC das características estruturais de um dispositivo e das características comportamentais do seu respectivo driver. A linguagem TDevC contém construções para: especificação da comunicação entre dispositivos, como protocolos de acesso a registradores; descrição de máquinas de estados refletindo o comportamento do dispositivo; e especificação das propriedades temporais que representam comportamentos específicos, que podem vir a ocorrer durante a execução do sistema.

Após a especificação do modelo na linguagem TDevC, a TDevCGen transforma o modelo descrito para um formato intermediário, montando na memória uma máquina de estados hierárquica, denominada HFSM-D.

Uma máquina de estados hieráquica, descrita em detalhes na seção 2.1, é uma máquina que representa o protocolo de comunicação entre dispositivos, refletindo exatamente a execução do sistema sob validação (cada estado está associado a um momento específico da execução). É importante destacar que cada estado pode conter um conjunto de propriedades temporais que aquele estado deve respeitar. Sendo que, além das propriedades locais, cada estado deve respeitar as propriedades do seu estado pai e dos seus ascendentes hierárquicos.

Com o formato intermediário em memória, a TDevCGen gera o monitor em C++ ou em SystemVerilog sintetizável. Na próxima etapa, de integração, esse novo componente é manualmente integrado na plataforma alvo. Nessa plataforma, os acessos a endereços de memória são interceptados e enviados a porta de observação do monitor. Assim, de posse dos dados trocados entre o dispositivo mestre e o periférico sob validação, o monitor analisa esses dados e, dependendo do tipo de acesso, do endereço relacionado e do valor lido ou atribuído, realiza uma transição de estado na máquina de estados hierárquica. Dessa forma, como a máquina de estados reflete o comportamento do dispositivo e associa as propriedades comportamentais aos estados, o monitor pode detectar o status do periférico e a ocorrência dos comportamentos descritos associados ao estado corrente.

Porém, para um adequado funcionamento do monitor, é preciso ter a garantia de que a máquina de estados hierárquica gerada pela especificação TDevC é correta, pois, caso contrário, existe a possibilidade de gerar um monitor inconsistente. Estamos assumindo a definição de um monitor inconsistente à possibilidade de ocorrência de dois comportamentos indesejados na máquina de estados hierárquica, são eles:

1. Estados com a presença de não-determinismo: um estado não-determinístico é aquele no qual para cada estados e símbolos de entrada pode haver vários próximos estados possíveis.

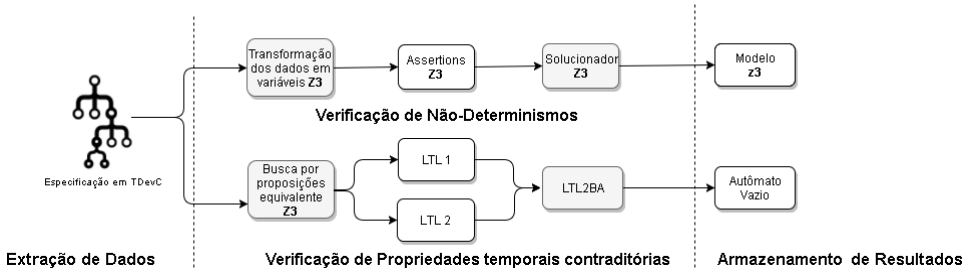
2. Existência de propriedades temporais contraditórias: uma máquina de estados hierárquica pode ter várias sequências de execução. A sequência escolhida irá depender do comportamento do driver durante a execução. Cada linha de execução possui uma sequência de propriedades temporais que devem ser satisfeitas, essas propriedades determinam o comportamento desejado. Porém, existe a possibilidade das propriedades se contradizerem

em algum momento da execução.

Caso o monitor gerado se deparasse com alguma desses comportamentos, seria preciso voltar para a etapa da especificação do modelo em TDevC e verificar se houve um erro na especificação ou um erro no modelo de referência.

Diante dos problemas apresentados, a proposta desse trabalho consiste na validação estática de uma especificação em TDevC para garantir que não seja gerado um monitor inconsistente de acordo com os aspectos mencionados. A figura 4.2 mostra uma visão geral das etapas da validação de uma especificação TDevC proposta nesse trabalho. No diagrama é possível verificar que as duas validações propostas (verificação de não-determinismos e verificação de propriedades temporais contraditórias) fazem uso do provador de teoremas Z3 RESEARCH (2016). Além disso, a validação que busca por propriedades temporais contraditórias faz uso, também, da ferramenta LTL2BA GASTIN; ODDOUX (2001), que transforma uma propriedade temporal em um autômato de Büchi e a partir dessa transformação verifica se existe contradição.

Figura 4.2: Visão geral das etapas da validação



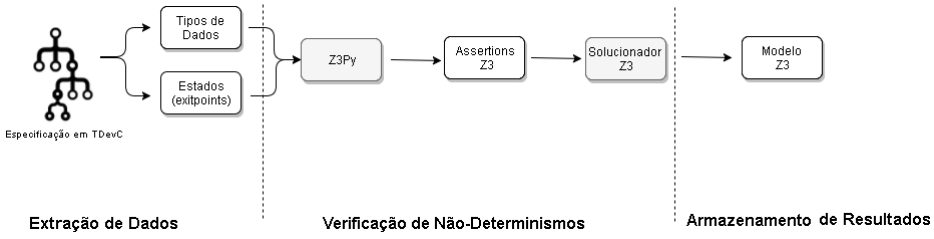
Nas próximas seções a proposta para a validação da máquina de estados hierárquica será apresentada em detalhes.

4.2 Checagem de não-determinismo na HFSM-D

Como mencionado na seção 2.1, a função de transição da HFSM-D é representada por proposições da lógica proposicional de primeira ordem. Essas transições podem ser formadas por proposições atômicas ou compostas. Como cada estado pode ter mais de uma transição de saída (mais de um *exitpoint*), para se ter uma máquina de estados determinística é preciso garantir que para cada par de estado e símbolo de entrada (representados por proposições simples ou compostas) exista apenas um próximo estado. Assim, será apresentada uma abordagem para verificar a existência de não-determinismos em uma especificação TDevC, a fim de gerar uma HFSM-D livre desse tipo de inconsistência.

A figura 4.3 mostra o fluxo da abordagem proposta. Primeiramente os dados são extraídos de uma especificação em TDevC. Após a extração, os dados são validados com o auxílio do provador de teoremas Z3. O resultado dessa validação é então armazenado. Para um melhor entendimento da proposta, nas próximas subseções a explicação será feita passo a passo, desde a extração dos dados de uma especificação TDevC até a checagem.

Figura 4.3: Diagrama do fluxo da checagem de não-determinismos em uma especificação TDevC



4.2.1 Extração dos dados de uma especificação em TDevC

A partir da especificação TDevC de um dispositivo, mostrada em detalhes na seção 2.3, os dados relevantes para a checagem de não-determinismos foram extraídos e agrupados seguindo um padrão. Esses dados padronizados serão posteriormente checados através da abordagem proposta. Para um melhor compreensão, a extração dos dados será dividida em três partes: na primeira parte os registradores, campos de registradores, variáveis, padrões e estados são especificados; na segunda parte, especifica-se todas as proposições atômicas que formarão os *exitpoints* de cada estado; e na terceira parte os estados, juntamente com os seus *exitpoints*, são declarados. A seguir, cada uma dessas partes são explicadas com mais detalhes.

4.2.1.1 Primeira parte da extração dos dados de uma especificação em TDevC

Como dito anteriormente na seção 2.1, os rótulos das transições da HFSM-D são formados por expressões da lógica proposicional. As variáveis dessas expressões podem ser formadas a partir de proposições atômicas. Essas proposições atômicas, por sua vez, podem ser formadas por registradores, referenciando os dados dos acessos realizados pela CPU à periféricos, variáveis, padrões e estados. Dessa forma, os primeiros dados extraídos da instância de uma especificação TDevC foram os registradores, os campos de registradores, as variáveis, os padrões e os estados.

Os dados extraídos foram agrupados seguindo o padrão mostrado na figura 4.4. Primeiramente na linha 1 tem-se a quantidade de registradores, campos de registradores, variáveis, padrões e estados que serão usados para formar as proposições atômicas. Os padrões dos tipos de dados são:

- Registrador. Esse tipo de dado é formado da seguinte forma: *reg* seguida pelo carácter %, pelo nome do registrador e por uma sequência de pontos que indica o tamanho do registrador em bits. Assim, nas linhas 3 e 4 tem-se a declaração dos registradores de nomes *REGISTRADOR_1* e *REGISTRADOR_2* de tamanho 8 bits, pois possuem 8 pontos;
- Campo de registrador. Esse tipo de dado é formado da seguinte forma: palavra *cmp%NomeDoRegistrador%Campo%NomeDoCampo*. Observe que o caractere % serve como separador de cada campo desse tipo. As linhas 4, 5 e 6 mostram essa construção, nas

quais os campos 2 e 3 do registrador *REGISTRADOR_1* são declarados com os nomes, respectivamente, *CAMPO_2* e *CAMPO_3*;

- Variável. Esse tipo de dados é formado da seguinte forma: palavra *var* seguida pelo carácter %, pelo nome da variável e por sua largura em bits. Na linha 7 a variável de nome *VARIAVEL_1* e de tamanho 8 bits é declarada.

- Padrão. Esse tipo de dados é formado da seguinte forma: palavra *pat* seguida pelo carácter %, pelo nome do padrão e pelo padrão. Na linha 8 tem-se a declaração do padrão de nome *PADRAO_1* cujo padrão é formado pelos 4 bits de ordem mais baixa com o valor zero (0) e o restante dos bits podendo assumir qualquer valor (.), zero (0) ou um (1);

- Estado. Esse tipo de dados é formado da seguinte forma: palavra *ste* seguida pelo carácter % e pelo nome do estado. A linha 9 mostra um exemplo, no qual o estado de nome *ESTADO_1* é declarado.

```
1 8
2 reg%REGISTRADOR_1 ........
3 reg%REGISTRADOR_2 ........
4 cmp%REGISTRADOR_1%2%CAMPO_2
5 cmp%REGISTRADOR_1%3%CAMPO_3
6 cmp%REGISTRADOR_2%1%CAMPO_1
7 var%VARIAVEL_1 ........
8 pat%PADRAO_1 ....0000
9 ste%ESTADO_1
```

Figura 4.4: Primeira parte da extração que checará a presença de não-determinismos

4.2.1.2 Segunda parte da extração dos dados de uma especificação em TDevC

A segunda parte dos dados extraídos de uma especificação em TDevC consiste das proposições atômicas que formarão os *exitpoints* de um determinado estado. Todos os registradores, campos, variáveis, padrões e estados utilizados devem ter sido declarados previamente na primeira parte da extração dos dados.

Os dados extraídos são agrupados seguindo o padrão apresentado na figura 4.5, no qual na primeira linha tem-se a quantidade de proposições atômicas que serão declaradas seguida por cada uma dessas proposições. É importante destacar as duas operações que podem ser realizadas nos registradores e campos de registradores: a leitura (*READ*) e a escrita (*WRITE*). Devido a natureza do meio de comunicação utilizado entre a CPU e os dispositivos, que é o barramento, apenas uma operação em apenas um registrador pode ser realizada por vez. Assim, em um mesmo momento, não é possível que haja uma leitura e uma escrita em diferentes registradores.

Caso haja uma operação de leitura ou escrita, o registrador ou campo do registrador é antecedido pelas palavras *READ* ou *WRITE*. As linhas 1, 2, 5, 6 e 7 da figura 4.5 mostram

exemplos dessas construções. Interpretando a proposição atômica mostrada na linha 1, por exemplo, significa dizer que houve uma escrita no registrador *REGISTRADOR_2*.

Nas linhas 3, 4, 8, 9 e 10 têm outros exemplos de declarações de proposições. Na linha 3, por exemplo, verifica-se se o valor do registrador *REGISTRADOR_2* segue o padrão definido em *PADRAO_1*. Já a proposição da linha 4 verifica se o valor do *REGISTRADOR_2* é igual a 32.

Durante a interpretação dos dados extraídos, as proposições declaradas são nomeadas internamente de $p0$ até pn, onde n é a quantidade de proposições declaradas menos 1, na sequência em que são declaradas. Logo, a primeira proposição atômica declarada passa a ser chamada de $p0$, a segunda de $p1$ e assim por diante, até a última proposição. O objetivo dessa nomeação é a simplificação da terceira parte da extração dos dados, a qual contém as proposições atômicas ou compostas de todas as transições de todos os estados da máquina.

```
1  10
2  WRITE%reg%REGISTRADOR_2
3  READ%reg%REGISTRADOR_2
4  reg%REGISTRADOR_2 == pat%PADRAO_1
5  reg%REGISTRADOR_2 == 32
6  READ%reg%REGISTRADOR_1
7  READ%cmp%REGISTRADOR_1%CAMPO_2
8  READ%cmp%REGISTRADOR_1%CAMPO_3
9  cmp%REGISTRADOR_1%CAMPO_2 == 1
10 cmp%REGISTRADOR_1%CAMPO_3 == 1
11 reg%REGISTRADOR_1 == var%VARIAVEL_1
```

Figura 4.5: Segunda parte da extração que checará a presença de não-determinismos

4.2.1.3 Terceira parte da extração dos dados de uma especificação em TDevC

Na terceira e última parte da extração dos dados, tem-se a declaração de cada um dos estados da máquina que será validada juntamente com os *exitpoints* que saem daquele estado. É importante destacar que nessa etapa os *entrypoints* já foram transformados em *exitpoints*, conforme explicado na seção 2.3.2.

```
1 2
2 State(Estado_1)
3 And(p1,p2) And(p1,p3)
4 State(Estado_2)
5 Or(And(p5,p7),And(p6,p8)) And(p4,p9)
```

Figura 4.6: Terceira parte da extração que checará a presença de não-determinismos

O padrão seguido nessa terceira parte da extração pode ser verificado na figura 4.6. Primeiramente, na linha 1, tem-se a quantidade de estados que serão declarados. Nas linhas subsequentes, tem-se a palavra *State* seguida entre parênteses pelo nome do estado e na próxima linha, os *exitpoints* do estado separados por um espaço em branco. Os *exitpoints* de cada estado

podem ser formados por uma proposição atômica ou por uma proposição composta formada por um dos cinco operadores abaixo:

1. $And(p,q)$: corresponde a conjunção das proposições p e q;
2. $Or(p,q)$: corresponde a disjunção das proposições p e q;
3. $Not(p)$: corresponde a negação da proposição p;
4. $Implies(p,q)$: corresponde a implicação da proposição p na proposição q; e
5. $eq(p,q)$: corresponde a equivalência das proposições p e q.

As proposições p e q podem ser proposições atômicas ou proposições compostas, formadas por um desses cinco operadores definidos.

As proposições são nomeadas de acordo com o padrão definido na segunda parte das extração dos dados. Na linha 3, da figura 4.6 tem-se os dois *exitpoints* do estado *ESTADO_1* e na linha 5, os dois *exitpoints* do estado *ESTADO_2*.

Assim, as três partes da extração dos dados de uma especificação TDevC ficará como mostrado na figura 4.7. São esses dados extraídos que serão validados segundo a técnica proposta.

```
1  8
2  reg%REGISTRADOR_1 ........
3  reg%REGISTRADOR_2 ........
4  cmp%REGISTRADOR_1%2%CAMPO_2
5  cmp%REGISTRADOR_1%3%CAMPO_3
6  cmp%REGISTRADOR_2%1%CAMPO_1
7  var%VARIAVEL_1 ........
8  pat%PADRAO_1 ....0000
9  est%ESTADO_1
10 10
11 WRITE%reg%REGISTRADOR_2
12 READ%reg%REGISTRADOR_2
13 reg%REGISTRADOR_2 == pat%PADRAO_1
14 reg%REGISTRADOR_2 == 32
15 READ%reg%REGISTRADOR_1
16 READ%cmp%REGISTRADOR_1%CAMPO_2
17 READ%cmp%REGISTRADOR_1%CAMPO_3
18 cmp%REGISTRADOR_1%CAMPO_2 == 1
19 cmp%REGISTRADOR_1%CAMPO_3 == 1
20 reg%REGISTRADOR_1 == var%VARIAVEL_1
21 2
22 State(Estado_1)
23 And(p1,p2) And(p1,p3)
24 State(Estado_2)
25 Or(And(p5,p7),And(p6,p8)) And(p4,p9)
```

Figura 4.7: Extração completa dos dados de uma especificação TDevC

4.2.2 Validação dos dados extraídos de uma especificação em TDevC

Para a implementação da técnica proposta para a validação de uma especificação em TDevC, foi utilizado o provador de teoremas Z3 (2.4). Primeiramente foi necessário definir os

tipos de dados que seriam utilizados. A API do Z3 em Python, Z3Py, possibilita o uso de vários tipos, entre eles podemos citar:

- Int(): A função Int('x') cria uma variável do tipo inteiro em Z3 com o nome x. Z3Py usa o símbolo '=' para atribuição e os operadores, <, <=, >, >=, == e != para comparação.
- Bool(): A função Bool('y') cria uma variável booleana com nome y. Z3Py suporta os seguintes operadores booleanos: And, Or, Not, Implies (Implicação), If (if-then-else) e == para equivalência.
- BitVec(): A função BitVec('x',t) cria um vetor de bits de nome x e com o tamanho t.
- Array(): Como parte da formulação de um programa da teoria da matemática computacional, McCarthy MCCARTHY (1996) propôs uma teoria básica de arrays, caracterizado pelos axiomas de seleção e armazenamento. A função Array('A',IntSort(),BitVecSort(t)) cria um array do tipo BitVec, ou seja, uma array de bits. A expressão Select(A,i) retorna o valor armazenado na posição i do array A e Store(A,i,v) retorna um novo array idêntico a A, mas na posição i é armazenado o valor v.

Baseado na definição das proposições atômicas apresentada na seção 2.1.1, como as variáveis e os registradores foram definidos como sendo pertencentes ao conjunto dos números inteiros, para facilitar a manipulaão, no provador de teoremas Z3 eles foram definidos como sendo um array de bits, o qual pode representar também um número inteiro. Os padrões foram definidos como sendo do mesmo tipo dos registradores. Como as operações de leitura (READ) e escrita (WRITE), assim como os estados, só podem assumir apenas dois valores (*true* ou *false*), eles foram declaradas como sendo do tipo $Bool()$.

A figura 4.8 mostra um pseudocódigo da abordagem utilizada para a validação. Após a definição dos tipos de dados, linha 3, e dos estados, linha 5, as proposições dos *exitpoints* de cada estado da máquina hierárquica são armazenadas em um vetor, conforme mostra no trecho de código das linhas de 8 a 10. Após o armazenamento dos *exitpoints*, cada um deles é verificado com todos os outros em busca de não-determinismo. Essa verificação consiste em adicionar, linha 14, cada par de *exitpoint* como uma restrição (ou *assertion*) em um solucionador Z3.

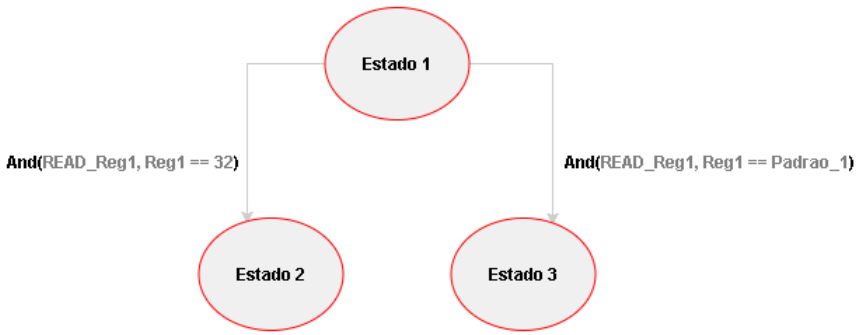
O Z3 fornece diferentes tipos de solucionadores. Para essa implementação, foi usado o solucionador básico. A função Solver(), linha 6, cria um solucionador de propósito geral. Restrições podem ser adicionadas nesse solucionador através do método add(), linha 15, e o método check(), linha 16, soluciona as restrições adicionadas. O Resultado é 'sat' se uma solução foi encontrada e 'unsat' se não existe solução. A solução encontrada é um modelo para o conjunto de restrições adicionadas no solucionador. Um modelo é uma interpretação que faz com que cada restrição adicionada seja valorada como verdadeira. Assim, se o resultado da checagem for 'sat', significa dizer que existe um modelo no qual os dois *exitpoints* adicionados ao solucionador são verdadeiros no mesmo momento, caracterizando o não-determinismo.

```
1  INICIO
2  VARIAVEIS
3  Registradores, Variaveis, Padroes, Estados: Variaveis Z3;
4  i,j,z:Inteiro;
5  estado: Estados da Especificacao TDevC
6  S:Solver();
7
8  PARA z de 1 ATE QUANTIDADE_DE_ESTADOS FACA PASSO 1
9      estado[z] <- EXITPOINTS[z];
10 FIM PARA PASSO 1;
11
12 PARA i de 1 ATE QUANTIDADE_DE_ESTADOS FACA PASSO 2
13     PARA j de 1 ATE QUANTIDADE_DE_EXITPOINTS FACA PASSO 3
14         SE i != j FACA PASSO 4
15       S.ADD() <- (estado[i][j],estado[i][j+1])
16       SE S.CHECK() IGUAL A SAT:
17       ESCREVER(S.MODEL());
18       FIM SE;
19         FIM PARA PASSO 4;
20   FIM PARA PASSO 3;
21 FIM PARA PASSO 2;
22 FIM
```

Figura 4.8: Pseudocódigo da abordagem proposta para a validação

Para uma ilustração dessa abordagem, tomemos como exemplo o trecho de uma máquina de estados mostrada na figura 4.9. Assumindo-se que os tipos de dados requeridos foram declarados e que o padrão **Padrao_1** é ..100000, o próximo passo no fluxo consiste em adicionar os dois *exitpoints* ao solucionador e checar se existe ou não um modelo. Para esse caso, o resultado da checagem foi *sat* e o modelo retornado é o apresentado na figura 4.10.

Figura 4.9: Exemplo de um estado com seus *exitpoints*



No modelo apresentado, na linha 1 tem-se o nome do estado que foi encontrado o não-determinismo, seguido nas linhas 3 e 4 pelos *exitpoints* que apresentaram o comportamento indesejado. Nas linhas 6, 7 e 8 tem-se os valores de *Reg1*, *Padrao_1* e *READ_Reg1* que fazem com que as duas transições de saída possam ser verdadeiras ao mesmo tempo. Na linha 6, é mostrado o valor 32 de *Reg1* em binário, apresentando o valor de cada campo separadamente. Na linha 8, a variável *READ_Reg1* possui seu valor *True* (Verdadeiro). Para que o *Padrao_1* seja igual a 32 e consequentemente o não-determinismo seja verificado, é necessário que as posições 6 e 7 do padrão *Padrao_1* possuam o valor zero (0), e como esse padrão é da forma ..100000,

significa que as posições 6 (.) e 7 (.) do padrão podem assumir qualquer valor. Assim, assumindo o valor zero (0), os dois *exitpoints* podem ser verdadeiros ao mesmo tempo, caracterizando o não-determinismo.

```
1 Estado 1
2 ##
3 And(READ_Reg1,Reg1 == Padrao_1)
4 And(READ_Reg1,Reg1 == 32)
5 ##
6 [Reg1 = [0 -> 0,1 -> 0,2 -> 0,3 -> 0,4 -> 0,5 -> 1,6 -> 0,7 -> 0],
7  Padrao_1 = [7 -> 0, 6 -> 0],
8  READ_Reg1 = True]
```

Figura 4.10: Modelo retornado para o exemplo mostrado na figura 4.9

No capítulo 5 serão apresentados os resultados dos experimentos realizado na especificação TDevC da *Ethernet DM9000A*.

4.3 Busca por contradições nas propriedades temporais de uma especificação TDevC

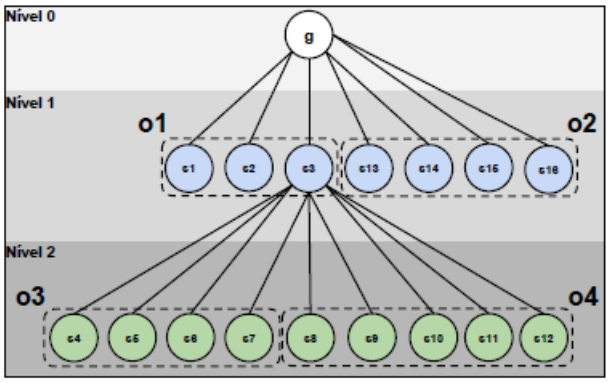
A linguagem da lógica temporal é uma ferramenta para a especificação e verificação de propriedades de sistemas reativos. Uma fórmula na lógica temporal, também conhecida como propriedade temporal, descreve o conjunto de sequências infinitas para as quais é verdadeira. Um determinado sistema satisfaz essa propriedade se todos os seus caminhos de execução pertencem a esse conjunto.

Um modelo da Lógica Temporal Linear (LTL) corresponde a uma sequência infinita de estados onde cada ponto no tempo tem um único sucessor. Uma fórmula p na lógica temporal, avaliada nessa sequência de estados, é satisfatível se existe uma sequência de estados S tal que $S \models p$ PNUELI; MANNA (1992).

No caso da máquina de estados hierárquica, modelada a partir da especificação TDevC, vimos que um determinado estado possui, além das propriedades declaradas, propriedades dos seus ascendentes hierárquicos, caso elas existam. Para deixar mais claro, transformando a figura 2.3, apresentada na seção 2.1, em uma visão simplificada dos níveis de hierarquia, figura 4.11, fica claro que as propriedades declaradas no estado global g (pai de todos os demais estados da máquina), devem ser respeitadas em todos os estados filhos de g, ou seja, os estados nos níveis 1 e 2. Enquanto que, as propriedades declaradas no estado $s3$ devem ser respeitadas em todos os seus estados filhos, nível 2. Assim, os estados filhos de $s3$ (de $s4$ à $s12$) devem respeitar as propriedades do estado $s3$ e as propriedades do estado global g.

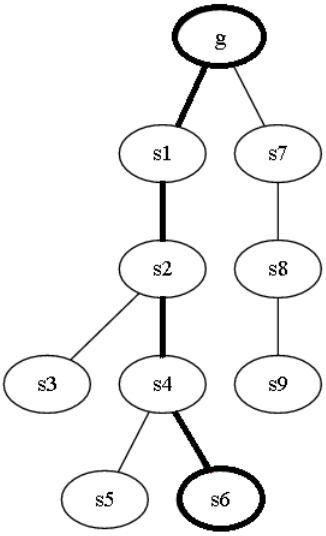
Atualmente, a validação das propriedades temporais dá-se apenas durante a execução da plataforma de checagem do driver do dispositivo. Dessa forma, durante a execução, o monitor MDDC pode deparar-se com a violação de uma propriedade, declarada de forma inconsistente,

Figura 4.11: Visão hierárquica da máquina mostrada na figura 2.3



que poderia ter sido evitada caso houvesse uma checagem estática prévia dessa propriedade. Além disso, essa propriedade inconsistente só seria descoberta se/quando o monitor alcançasse o estado ao qual ela pertence. Por exemplo, na máquina de estados mostrada na figura 4.12, caso duas propriedades contraditórias fossem declaradas no estado g, com certeza o monitor iria detectar prontamente a violação de uma das propriedades, uma vez que essas propriedades são checadas desde o início do monitoramento.

Figura 4.12: Máquina de estados com destaque para a linha de execução



Por outro lado, caso exista uma propriedade no estado $s6$ contraditória à uma propriedade do estado g, o monitor só iria detectar a violação dessa propriedade caso seguisse pela computação, ou linha de execução, destacada na figura. Como a linha de execução seguida depende do comportamento do driver, não seria possível garantir que o monitor iria detectar a violação da propriedade, podendo detectar apenas através de um conjunto de execuções (testes). Além disso, tanto no primeiro caso quanto no segundo, o monitor apenas irá possuir a informação da violação de uma propriedade, não sabendo que existem propriedades declaradas de forma contraditória e nem quais seriam essas propriedades.

Baseado nisso, é importante haver uma checagem dessas propriedades previamente

antes da geração do monitor MDDC, pois, caso contrário, propriedades inconsistentes somente poderiam ser encontradas durante a execução da plataforma. Para evitar que propriedades sejam violadas em virtude de especificações contraditórias, esse trabalho propõe, também, uma prévia validação da HFSM-D em busca dessas propriedades.

4.3.1 Extração dos dados de uma especificação em TDevC

A partir da especificação TDevC de um dispositivo, mostrada em detalhes na seção 2.3, os dados necessários para a busca por propriedades temporais contraditórias foram extraídos. Assim como na extração dos dados para a verificação de estados não-determinísticos, o resultado da extração é composto por três partes: declaração dos registradores, campos, variáveis, padrões e estados utilizados; declaração das proposições atômicas usadas na especificação das propriedades temporais; e finalmente a declaração das propriedades temporais atreladas ao estado que pertencem.

A primeira e segunda parte dessa extração segue o mesmo padrão da primeira (figura 4.4) e da segunda (figura 4.4) parte dos dados extraídos para a verificação de estados não-determinísticos.

A terceira parte dos dados extraídos, porém, segue o padrão mostrado no exemplo da figura 4.13. Na primeira linha tem-se a quantidade de estados que serão especificados. As próximas linhas seguem o seguinte padrão:

$$(.)^*(NomeDoEstado)^+(\%NomeDaFormulaLTL\%FormulaLTL)^*$$

$(.)^*$ indica que podem ter zero ou mais pontos antecedendo o nome do estado. A quantidade de pontos indica o nível no qual o estado se encontra. Assim, o estado global não terá nenhum ponto antecedendo o seu nome, já que está no nível zero. Os estados pertencentes ao estado global, porém, como estão no nível um da hierarquia, terão um ponto antecedendo o seu nome e assim por diante. $(NomeDoEstado)^+$ indica que o nome do estado deve aparecer pelo menos uma vez. E $(\%NomeDaFormulaLTL\%FormulaLTL)^*$ indica que o nome do estado pode ser seguido de uma ou mais fórmulas LTLs, separadas pelo caractere %.

```
1 4
2 EstadoGlobal%LTL1%[](q0 && q3)%LTL2%<>q5
3 .EstadoFilho1
4 .EstadoFilho2%LTL3%q0 U q3%LTL4%[]!q5
5 ..EstadoNeto1
```

Figura 4.13: Terceira parte da entrada da aplicação que checa a existência de propriedades temporais contraditórias

Assim, na linha 2 da figura 4.13 tem-se a declaração do estado global seguida da declaração de duas fórmulas LTLs de nomes LTL1 e LTL2 e fórmulas, respectivamente,[](q0 && q3) e <>q5. Nas linhas 3 e 4, tem-se os dois estados filhos do estado global e na linha 4 tem-se a declaração de um estado filho do estado de nome EstadoFilho2.

A figura 4.14 mostra como ficaria uma entrada completa.

```
8
reg%REGISTRADOR_1 ........
reg%REGISTRADOR_2 ........
cmp%REGISTRADOR_1%2%CAMPO_2
cmp%REGISTRADOR_1%3%CAMPO_3
cmp%REGISTRADOR_2%1%CAMPO_1
var%VARIAVEL_1 ........
pat%PADRAO_1 ....0000
est%ESTADO_1
10
WRITE%reg%REGISTRADOR_2
READ%reg%REGISTRADOR_2
reg%REGISTRADOR_2 == pat%PADRAO_1
reg%REGISTRADOR_2 == 32
READ%reg%REGISTRADOR_1
READ%cmp%REGISTRADOR_1%CAMPO_2
READ%cmp%REGISTRADOR_1%CAMPO_3
cmp%REGISTRADOR_1%CAMPO_2 == 1
cmp%REGISTRADOR_1%CAMPO_3 == 1
reg%REGISTRADOR_1 == var%VARIAVEL_1
4
EstadoGlobal%LTL1%[](q0 && q3)%LTL2%<>q5
.EstadoFilho1
.EstadoFilho2%LTL3%q0 U q3%LTL4%[]!q5
..EstadoNeto1
```

Figura 4.14: Entrada completa da aplicação que checa a existência de propriedades temporais contraditórias

4.3.2 Técnica utilizada na análise dos dados extraídos

Existe uma ligação íntima entre modelos de fórmulas LTL e linguagens de palavras infinitas: o modelo de uma fórmula LTL constitui uma linguagem ω-regular sobre um alfabeto apropriado. Como resultado, o problema da satisfatibilidade para fórmulas LTL se reduz a checagem da vacuidade em linguagens ω-regular MUKUND (1997). Essa conexão foi explicitada pela primeira vez em SISTLA; VARDI; WOLPER (1983).

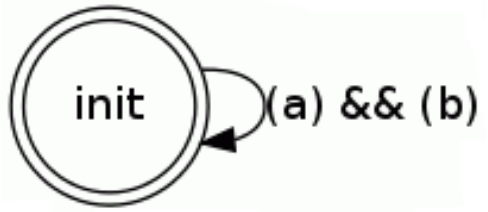
As linguagens ω-regulares são uma classe das linguagens-ω que generalizam a definição de linguagens regulares para palavras infinitas. Uma linguagem regular é, por sua vez, uma linguagem formal que pode ser expressa usando expressões regulares, ou seja, é uma linguagem produzida usando as operações de concatenação, união e fecho de Kleene sobre os elementos de um alfabeto.

Autômatos de Büchi (seção 2.2.4) são os autômatos que aceitam as linguagens ω-regulares e todas as linguagens ω-regulares são reconhecíveis por um autômato de Büchi. Dessa forma, é possível transformar uma fórmula LTL em um autômato de Büchi. Existem vários algoritmos e ferramentas que implementam esses algoritmos de tradução de uma fórmula LTL em um autômato de Büchi equivalente SISTLA; VARDI; WOLPER (1983)GERTH et al. (1995). Essa transformação é normalmente realizada em dois passos. No primeiro passo, um autômato de Büchi generalizado (GBA) é produzido a partir da fórmula e no segundo passo esse GBA é traduzido em um BA.

Figura 4.15: Autômato de Büchi retornado pela ferramenta Spin para a fórmula LTL [](a && b)

```
accept_init :
T0_init :
    if
    :: ((a) && (b)) -> goto T0_init
    fi;
```

(a) Código

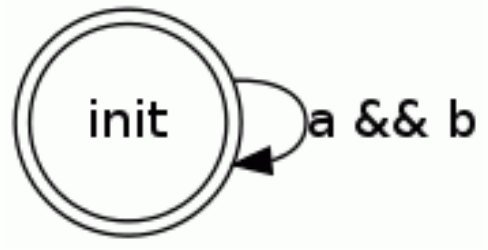


(b) Representação gráfica

Figura 4.16: Autômato de Büchi retornado pela ferramenta LTL2BA para a fórmula LTL [](a && b)

```
accept_init :
    if
    :: (a && b) -> goto accept_init
    fi;
```

(a) Código



(b) Representação gráfica

A ferramenta Spin HOLZMANN (1997) e LTL2BA GASTIN; ODDOUX (2001), por exemplo, realizam essa transformação recebendo como entrada uma fórmula LTL e retornando como saída a representação do autômato de Büchi em um formato de texto. Por exemplo, para a fórmula LTL [](a && b) a figura 4.15 mostra o autômato retornado pela ferramenta Spin e a figura 4.16 o autômato retornado pela ferramenta LTL2BA. Na parte a das figuras, é mostrado o autômato em um formato de texto e na parte b em um formato gráfico (para uma melhor visualização). Pela figura, percebe-se que ambos os autômatos possuem um estado de aceitação, indicando que a linguagem do autômato não é vazia.

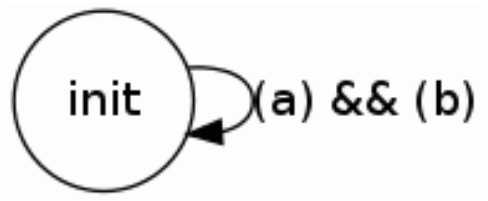
Como visto anteriormente, o problema da satisfatibilidade para fórmulas LTL se reduz a checagem da vacuidade em linguagens ω-regular. Assim, se o autômato de Büchi resultante da transformação de uma fórmula LTL possui uma linguagem vazia, significa dizer que a fórmula LTL não é satisfatível. O autômato de Büchi em formato de texto retornado pelas duas ferramentas diferem quando ele não possui nenhum estado de aceitação. A ferramenta LTL2BA retorna um autômato simplificado, enquanto a ferramenta Spin não.

Por exemplo, considerando a fórmula LTL [](a && b) && <> !b, pode-se verificar que ela não é satisfatível, pois não existe um modelo que satisfaça sempre (operador []) a *e* (operador &&) b e eventualmente no futuro (operador <>) satisfaça a negação de b (!b). Assim, o autômato de Büchi retornado para essa fórmula é um autômato vazio, sem nenhum estado de aceitação. O autômato retornado pela ferramenta Spin é o apresentado na figura 4.17 e o retornado pela ferramenta LTL2BA é o mostrado na figura 4.18. É possível perceber porém, que o autômato

Figura 4.17: Autômato de Büchi retornado pela ferramenta Spin para a fórmula LTL [](a && b) && <> !b

```
T0_init :
    if
    :: ((a) && (b)) -> goto T0_init
    fi;
```

(a) Código

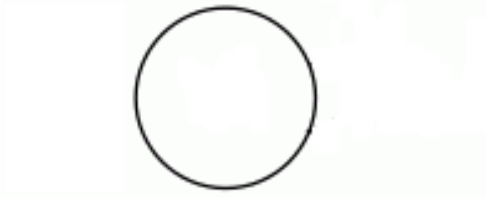


(b) Representação gráfica

Figura 4.18: Autômato de Büchi retornado pela ferramenta LTL2BA para a fórmula LTL [](a && b) && <> !b

```
false;
```

(a) Código



(b) Representação gráfica

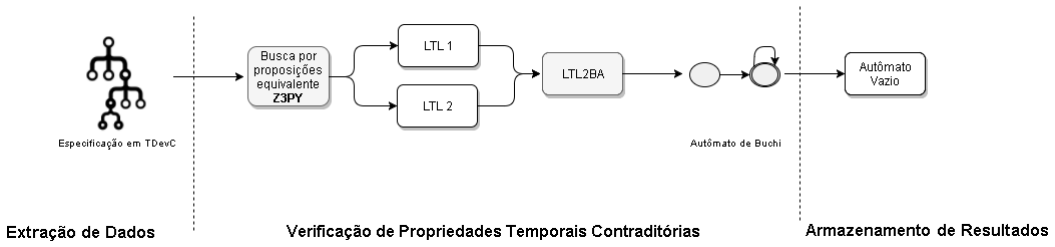
retornado pela ferramenta Spin, figura 4.17-a, apesar de não possuir nenhum estado de aceitação, não é de fácil interpretação quando comparado com o retornado pela ferramenta LTL2BA, figura 4.18-a, que retorna a palavra *false* para indicar que a linguagem do autômato é vazia.

Dessa forma, por retornar um autômato de Büchi simplificado, a ferramenta utilizada nesse trabalho foi a LTL2BA.

4.3.3 Validação dos dados extraídos de uma especificação em TDevC

A abordagem utilizada nesse trabalho para verificar se existem contradições entre as fórmulas LTLs pode ser verificada no diagrama mostrado na figura 4.19. Após a extração dos dados de uma especificação TDevC, o primeiro passo realizado é a checagem de equivalências entre as proposições atômicas que formam as fórmulas LTLs. Essa checagem, realizada com o auxílio do provador de teoremas Z3, permite que contradições entre as propriedades temporais não sejam mascaradas, nomeando de forma igual as proposições equivalentes.

Figura 4.19: Diagrama da abordagem utilizada para a verificação de contradições entre as propriedades temporais



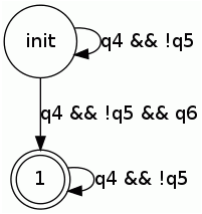
O segundo passo, após a nomeação única das proposições equivalentes, consiste em validar todas propriedades temporais de cada estado e dos seus ascendentes hierárquicos. Para isso, realiza-se uma conjunção (and lógico) a cada duas fórmulas LTLs e transforma-se essa nova fórmula em um autômato de Büchi. Caso o autômato retornado possua uma linguagem vazia, significa que a fórmula LTL resultante da conjunção é insatisfatível. Consequentemente, não existe um conjunto de infinitas sequências que satisfaça as duas fórmulas LTL ao mesmo tempo, caracterizando uma contradição entre elas.

Para deixar mais clara a abordagem proposta, vejamos o exemplo que se segue. Sendo:

- LTL 1: [](q1 && !q2) - (Sempre $q1$ e não $q2$)
- LTL 2: <>(q1 && q3) - (Eventualmente no futuro $q1$ e $q3$)

O autômato gerado da conjunção dessas duas propriedades é o mostrado na figura 4.20. Nele, é possível verificar a existência de um estado de aceitação, indicando que a conjunção da LTL 1 com a LTL2 é satistatível.

Figura 4.20: Autômato gerado da conjunção das LTLs 1 e 2



Esse resultado, porém, pode ser um falso positivo caso não haja a verificação prévia das proposições equivalentes. A tabela 4.1 mostra o valor de cada uma das proposições que formam as duas fórmulas LTLs.

Tabela 4.1: Proposições atômicas

Proposição	Conteúdo
$q1$	a == 1
$q2$	x == (y + 1)
$q3$	(x - 1) == y

Da tabela, temos $q2$ e $q3$ são proposições equivalentes, pois isolando a variável x em $q3$ temos que $q3$ ficará igual x == (y + 1), que é exatamente $q2$. Assim, reescrevendo as duas fórmulas LTLs, atribuindo a mesma nomenclatura para $q2$ e $q3$ temos:

- LTL 1: [](q1 && !q) - (Sempre $q1$ e não q)
- LTL 2: <>(q1 && q) - (Eventualmente no futuro $q1$ e q)

Gerando o autômato de Büchi da conjunção dessas duas novas fórmulas, o resultado é um autômato vazio, sem nenhum estado de aceitação, caracterizando a contradição, pois para a LTL

1 ser valorada como verdadeira é necessário que $q1$ seja sempre verdadeira e q sempre falsa, já para a LTL 2 ser valorada como verdadeira é necessário que em algum momento $q1$ **e** q sejam verdadeiras. Logo, não é possível ter um modelo no qual q seja sempre falsa, e eventualmente verdadeira ao mesmo tempo.

No capítulo 5 serão apresentados os resultados dos experimentos realizado na especificação TDevC da *Ethernet DM9000A*.

5

Experimentos e Resultados

Este capítulo descreve os experimentos realizados para a validação da proposta dessa dissertação, que consiste na validação de uma especificação TDevC para o desenvolvimento de *device drivers* robustos. Os experimentos foram realizados com base no dispositivo controlador *Ethernet DM9000A* DAVICOM (2006).

5.1 Experimento realizado

Os experimentos realizados com esse controlador cobriram os serviços da camada física (*PHY*), tais como: ligar, desligar, *reset*, transmissão e recepção de dados, definição do modo de operação e monitoramento *link status*. Para atender a esse propósito, a especificação TDevC desse controlador foi construída com 56 registradores e 15 estados. A figura 5.1 mostra uma visão geral da máquina de estados hierárquica gerada através da especificação TDevC. Para uma melhor visualização, as condições das transições de saída de cada estado foram omitidas e as propriedades LTL presentes estão simbolicamente representadas por LTL*n* onde *n* vai de 1 a 9.

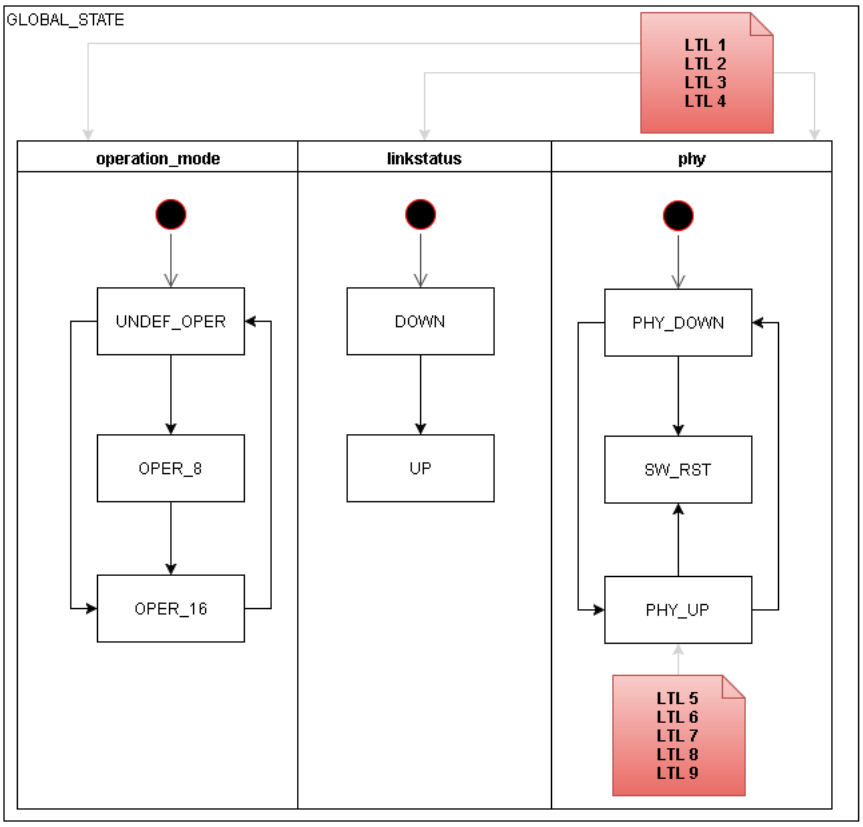
Da figura 5.1 temos que o estado global (*Global_State*) é composto por três regiões ortogonais:

1. *operation_mode*: essa região descreve a escolha do modo de operação entre 8 e 16 bits. O modo de operação é definido quando o *driver* requisita essa informação do dispositivo. Enquanto o *driver* não efetua essa leitura, o modo de operação permanece indefinido (permanecendo no estado inicial dessa região ortogonal, o *UNDEF_OPER*). É o modo de operação que irá ditar como o envio e o recebimento dos dados será realizado.

2. *linkstatus*: essa região especifica quando um link (conexão) está ativo.

3. *phy*: região ortogonal onde ocorre o controle da camada física. O estado inicial dessa região, *PHY_DOWN*, indica que inicialmente a camada está desligada. O estado *PHY_UP*, significa que a camada encontra-se ligada e pronta para transmitir e receber, enquanto que o estado *SW_RST* indica que o dispositivo está em um estado de *reset* solicitado pelo software.

Figura 5.1: Visão geral da HFSM-D da Ethernet DM9000A



As propriedades temporais foram definidas com base no *datasheet* para garantir que o dispositivo fosse controlado corretamente. Foram definidas quatro propriedades temporais no estado global, as quais devem ser respeitas em todos os estados da máquina hierárquica. A mais crítica dessas propriedades é a propriedade definida para garantir que a camada física jamais fosse ligada sem a definição do modo de operação.

O estado *PHY_UP*, por sua vez, possui outras três regiões ortogonais, conforme mostra a imagem 5.2. A região *config_data_transmission* é responsável pela escrita da carga útil e pela configuração do pacote a ser transmitido; a região *send_data* é responsável pelo envio físico do pacote; e a região *receive_data* é responsável pelo recebimento físico do pacote.

Outras cinco propriedades foram adicionadas no estado *PHY_UP*. Dessa forma, essas propriedades deverão ser respeitadas em todos os estados pertencentes ao estado *PHY_UP*. Três dessas propriedades merecem destaque. A primeira propriedade diz que enquanto um pacote está transmitindo (estado *SEND_DATA* da região ortogonal *send_data*), a configuração de novo dado não pode modificar o seu tamanho (estado *SET_LENGTH* da região ortogonal *config_data_transmission*). A outra propriedade é com relação ao tamanho do pacote escrito pela CPU e o tamanho informado. A CPU deve especificar corretamente o tamanho do pacote igual à quantidade de bytes escritos. A última propriedade que merece destaque, diz respeito, também, à quantidade de bytes, porém dos pacotes recebidos. A tabela 5.2 lista todas as propriedades definidas no estado *GLOBAL_STATE* e no estado *PHY_UP*.

Figura 5.2: Visão geral da HFSM-D da Ethernet DM9000A com o estado *UP* expandido

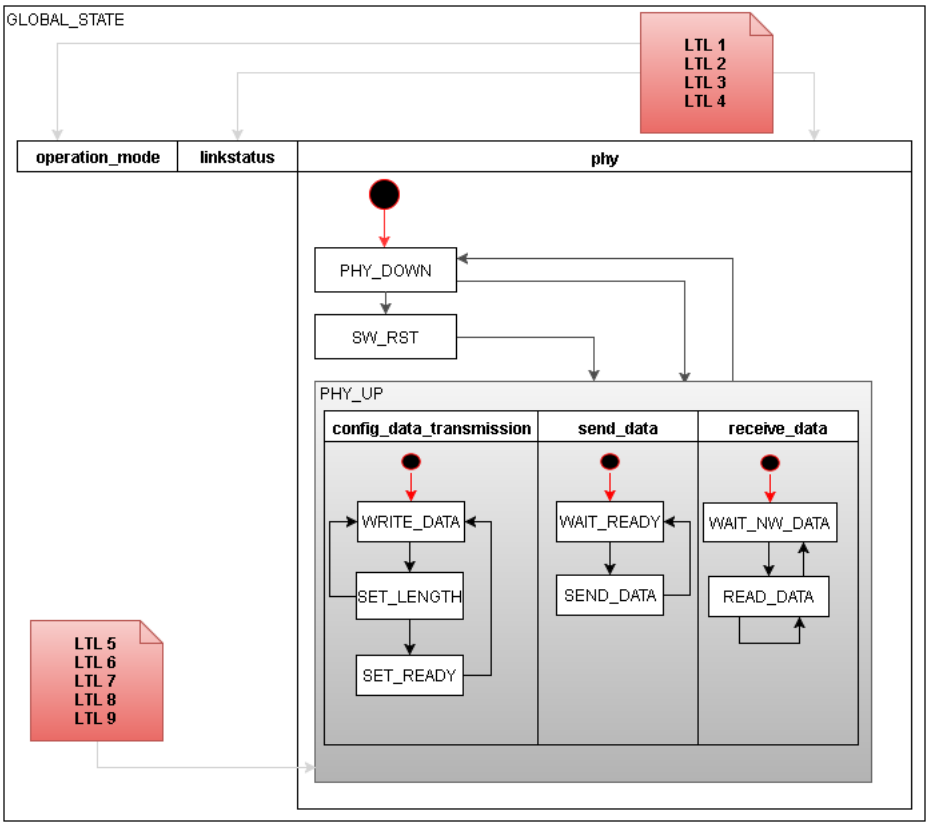


Tabela 5.1: Propriedades temporais definidas para a Ethernet DM9000A

Propriedade	Nome	Fórmula
LTL 1	rxtxinterruption	[]((q11 -> (!(q3 && q16 && q17) -> !(q18 || q19)))
LTL 2	UndefOperMode	[]((q11 -> !q0)
LTL 3	ForceReset	[]((q13 -> q14)
LTL 4	PromiscuousMode	[]((q12)
LTL 5	TXLenPPT	[]((q4 -> ((q6 -> ((q7) || (q8))) && (q9 -> (q7))))
LTL 6	WriteBeforeLen	[]((q1 || q2) -> !q3)
LTL 7	LenBeforeSend	!q4 U q5
LTL 8	UndefinedOperMode	[]((!q0)
LTL 9	NeverLenAndSend	[]((!(q4 && q5))

5.2 Métricas Avaliadas

Uma das métricas avaliadas no experimento foi o tempo de execução ou latência, que diz respeito ao intervalo de tempo decorrido entre o início e o fim da validação. Apesar de ser um tempo relativo à plataforma utilizado no experimento, ele pode ser usado como base para a comparação de aplicações que executem em uma plataforma semelhante. Os resultados são apresentados em segundos. A outra métrica avaliada foi um contador de comportamentos indesejados, que corresponde à quantidade de estados onde foram encontrados o não-determinismo e à

Tabela 5.2: Proposições atômicas da tabela 5.2

q0	UNDEF_OPER
q1	WRITE_MDRALR
q2	WRITE_MDRAHR
q3	WRITE_DATA
q4	SEND_DATA
q5	SET_LENGTH
q6	OPER_16
q7	txlen == txpkgcounter
q8	txlen == txpkgcounter - 1
q9	OPER_8
q11	PHY_UP
q12	RCR_PRMSC != 1
q13	READ_DATA
q14	READ_MRCMD
q15	UP
q16	WAIT_READY
q16	WAIT_NW_DATA
q18	BMCR_POWERDN == 1
q19	GPR_PHYPD == 1

quantidade de propriedades temporais contraditórias.

5.3 Resultados

A especificação TDevC do dispositivo controlador *Ethernet DM9000A* foi validada através da abordagem proposta nesse trabalho. Como resultado, não foram encontrados estados com a presença de não-determinismos, tampouco contradições entre as propriedades temporais foram encontradas. A latência desse primeiro experimento pode ser visualizada na tabela 5.3

Tabela 5.3: Resultado do primeiro experimento

Checagem	**Latência**	**Casos de comportamento indesejado**	**Ambiente**
Não-determinismo	18	0	Processador Intel Core i7, 2.10 GHz, 6 GB RAM
Propriedades contraditórias	0.37	0	Processador Intel Core i5, 2.5 GHz, 4 GB RAM

Assim, o projetista da especificação TDevC inseriu nessa especificação, de maneira imparcial, possíveis casos de não-determinismos e contradições entre as propriedades temporais. Dessa forma, não seria possível saber previamente onde esses comportamentos indesejados foram inseridos, para não gerar resultados tendenciosos. Assim, uma nova checagem foi realizada e

todos os comportamentos indesejados que foram inseridos foram reportadas pelas validações propostas nessa dissertação.

5.3.1 Resultados da técnica que verifica a presença de não-determinismos

A técnica que checa a presença de não-determinismos na especificação TDevC mostrou-se eficaz no experimento realizado pois foi capaz de encontrar todos os não-determinismos inseridos na especificação. Os casos de não-determinismo encontrados para a entrada gerada podem ser visualizados na tabela 5.4:

Tabela 5.4: Não-determinismos encontrados no segundo experimento

Estado	**Transição 1**	**Transição 2**	**Contra-exemplo**
PHYDN	And(WRITE_BMCR_SPEEDSEL, BMCR[4] == 0)	And(WRITE_BMCR_POWER_DN, BMCR[2] == 0)	BMCR_POWER_DN = 0; BMCR = [2 -> 0, 4 -> 0, else -> 0]; WRITE_BMCR_POWER_DN = True; WRITE_BMCR_SPEEDSEL = True
TXSDPKGIDLE	And(WRITE_CHIPR, CHIPR == RXNOERROR)	And(WRITE_CHIPR, CHIPR == 32)	CHIPR = [0 -> 0,1 -> 0,2 -> 0, 3 -> 0,4 -> 0,5 -> 1,6 -> 0,7 -> 0]; RXNOERROR = [5 -> 1, 4 -> 0, 7 -> 0, 6 -> 0]; WRITE_CHIPR = True
TXSDPKGSDING	And(READ_NSR_TX1END, NSR == t1)	Or(And(READ_NSR_TX1END, NSR_TX1END == 1), And(READ_NSR_TX2END, NSR_TX2END == 1))	READ_NSR = True; READ_NSR_TX2END = True; NSR = [3 -> 1, else -> 1], t1 = [3 -> 1, else -> 1]

Na primeira coluna tem-se o nome do estado que foi encontrado o não-determinismo. Na segunda e terceira colunas tem-se os *exitpoints*, ou transições de saída, responsáveis pelo comportamento indesejado. E na última coluna é apresentado um contra-exemplo, ou um modelo, retornado pelo solucionador, que faz com que os dois *exitpoints* sejam valorados como verdadeiros, caracterizando o não-determinismo.

5.3.2 Resultados da técnica que verifica a presença de propriedades temporais contraditórias

A estratégia proposta, que checa a presença de propriedades temporais contraditórias, foi capaz de encontrar as duas contradições inseridas pelo projetista na especificação TDevC. Como mostra a tabela 5.5, a primeira contradição encontrada foi entre uma propriedade do estado global (*Global_State*) e outra do estado *PHYUP*. Já a outra contradição foi entre duas propriedades do estado *PHYUP*. Na tabela, $p1$ e $p2$ referem-se as propriedades contraditórias.

Tabela 5.5: Propriedades contraditórias encontradas no segundo experimento

Estado de $p1$	$p1$	**Estado de** $p2$	$p2$
Global_State	<>(q1 && q0)	*PHYUP*	[](!q0)
PHYUP	!q3 U q4	*PHYUP*	[]!(q3 || q4)

Analisando o resultado mostrado na tabela 5.5, tem-se que o autômato retornado da conjunção das propriedades $p1$ e $p2$, nos dois casos apresentados, é vazio, significando uma

linguagem vazia, e consequentemente, a inexistência de um modelo que satisfaça as duas propriedades ao mesmo tempo. A propriedade <>(q1 && q0) será valorada como verdadeira se em algum momento $q1$ e $q0$ forem valorados como verdadeiros simultaneamente. Já a propriedade [](!$q0$) será valorada como verdadeira se sempre o valor de $q0$ for valorado como falso. Assim, como a primeira propriedade requer que o valor de $q0$ seja verdadeiro e a segunda requer que ele seja falso, não existe um modelo que satisfaça essas duas propriedades ao mesmo tempo.

Analisando a segunda contradição encontrada temos: o operador da lógica temporal U (até que), conforme apresentado na seção 2.2, é interpretado da seguinte maneira na fórmula !q3 U q4: !q3 deve ser verdadeiro até que q4 seja verdadeiro. Portanto, inicialmente temos que !$q3$ deve ser valorado como verdadeiro (logo, $q3$ será falso) e logo após $q4$ será valorado como verdadeiro. A outra fórmula ([]!(q3 || q4)), que pode ser transformada em [](!q3 && !q4) pela lei de De Morgan, diz que sempre !$q3$ e !$q4$ deverão ter valores verdadeiros, o que implica que $q3$ e $q4$ devem ser sempre falsos para que a fórmula seja valorada como verdadeira. Assim, se a primeira fórmula requer que $q4$ seja valorada com o valor verdadeiro e a segunda com o valor falso, estamos diante de uma contradição.

A tabela 5.6 mostra os resultados desse segundo experimento. O aumento da latência justifica-se pelo fato de os resultados da checagem estarem sendo escritos em um arquivo externo apenas quando as inconsistências são encontradas, além do maior processamento em virtude dos contra-exemplos retornados.

Tabela 5.6: Resultado do segundo experimento

Checagem	**Latência**	**Casos de comportamento indesejado**	**Ambiente**
Não-determinismo	45	3 estados	Processador Intel Core i7, 2.10 GHz, 6 GB RAM
Propriedades contraditórias	0.44	4 propriedades	Processador Intel Core i5, 2.5 GHz, 4 GB RAM

5.4 Conclusão

Aplicando a abordagem proposta nesse trabalho em um exemplo prático foi possível testar a sua eficácia. A partir dos experimentos realizados, pode-se concluir que a abordagem proposta nesse trabalho mostrou-se satisfatória, uma vez que foi capaz de validar a especificação TDevC de um dispositivo.

6

Conclusão

A linguagem TDevC é uma linguagem de domínio específico (DSL), que objetiva dar suporte à especificação do comportamento do protocolo de alto nível de comunicação entre um dispositivo e o seu driver, e de assertivas relacionadas a esse comportamento através de propriedades temporais(LTL). Essa linguagem permite especificar elementos estruturais, protocolos de acesso à memória dos dispositivos e o comportamento ideal para o uso destes periféricos. Esse comportamento é representado através de uma máquina de estados hierárquica (HFSM-D). A especificação TDevC dessa HFSM-D pode, no entanto, conter inconsistências.

Assim, o presente trabalho propôs uma abordagem para encontrar não-determinismos e propriedades temporais contraditórias nessa máquina de hierárquica. A busca por não determinismos foi realizada através da checagem, usando o provador de teoremas Z3, das condições das transições de saída dos estados da máquina. Caso exista a possibilidade de algumas dessas transições serem verdadeiras ao mesmo tempo, o não-determinismo é reportado.

A checagem das propriedades foi realizada usando a ferramenta LTL2BA apresentada em GASTIN; ODDOUX (2001) que transforma uma fórmula LTL em um autômato de Büchi. As propriedades de cada estado e dos seus ascendentes hierárquicos foram testadas duas a duas em busca de contradições. Como resultado, caso uma contradição seja encontrada, os estados envolvidos, assim como as propriedades contraditórias, são reportados.

Os experimentos e resultados, realizados em um controlador Ethernet DM9000A, mostraram que as abordagens propostas nessa dissertação foram capazes de serem testadas em um exemplo real e que 100% das inconsistências inseridas de maneira imparcial foram detectadas. Comparando com os trabalhos apresentados em SIMS; BUTTS; RANVILLE (2002) e TOYN; GALLOWAY (2007), percebe-se que a abordagem proposta nesse trabalho é mais completa, pois possibilita usar tipos de dados mais complexos, como registradores e padrões, na checagem de estados não-determinísticos.

Como trabalho futuro, propõe-se a integração do presente trabalho à plataforma apresentada em MACIEIRA; BARROS; ASCENDINA (2014), além de reportar a causa principal que originou a contradição, encontrada na checagem de propriedades temporais contraditórias, e não apenas relacionar as propriedades contraditórias com os estados aos quais pertencem.

Referências

BHARADWAJ, R.; SIMS, S. Salsa: combining constraint solvers with bdds for automatic invariant checking. In: TOOLS AND ALGORITHMS FOR THE CONSTRUCTION AND ANALYSIS OF SYSTEMS, Paris, France. **Anais...** Springer Berlin Heidelberg, 2000. p.378–395. (Lecture Notes in Computer Science, v.2102).

BJORNER, N.; MOURA, L. de. Z3: an efficient smt solver. **Springer**, [S.l.], 2008.

BJORNER, N.; MOURA, L. de. Z3: applications, enablers, challenges and directions. **Sixth International Workshop on Constraints in Formal Verification Grenoble**, [S.l.], 2009.

CHOU, A. et al. **An empirical study of operating systems errors**. [S.l.]: ACM, 2001. v.35, n.5.

CORBET, J.; RUBINI, A.; KROAH-HARTMAN, G. **Linux Device Drivers, Third Edition**. [S.l.]: OReilly Media, Inc, 2005.

DAVICOM. **DM9000A Ethernet Controller with General Processor Interface - Datasheet**. [S.l.]: Davicom Semiconductor, 2006. (DM9000A-17-DS-F01).

ECKER, W.; MüLLER, W.; DöMER, R. Hardware-dependent Software Principles and Practice. **Springer**, [S.l.], 2009.

FELTY, A. P.; NAMJOSHI, K. S. Feature Specification and Automated Conflict Detection. **ACM Transactions on Software Engineering and Methodology, Vol. 12, No. 1, Pages 3-27**, [S.l.], 2013.

GASTIN, P.; ODDOUX, D. Fast LTL to Büchi Automata Translation. In: PROCEEDINGS OF THE 13TH INTERNATIONAL CONFERENCE ON COMPUTER AIDED VERIFICATION (CAV'01), Paris, France. **Anais...** Springer, 2001. p.53–65. (Lecture Notes in Computer Science, v.2102).

GERTH, R. et al. Simple On-The-Fly Automatic Verification of Linear Temporal Logic. **Proceedings of the Fifteenth IFIP WG6.1 International Symposium on Protocol Specification, Testing and Verification XV**, [S.l.], p.3–18, 1995.

H., H. R.; Z., H.; KURSHAN, R. P. COSPAN. **Springer-Verlag New York**, [S.l.], 1996.

HOLZMANN, G. J. The model checker SPIN. **IEEE Transactions on software engineering**, [S.l.], v.23, n.5, p.279, 1997.

HOPCROFT, J. E.; MOTWANI, R.; ULLMAN, J. D. **Introduction to Automata Theory, Languages, and Computation (3rd Edition)**. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2006.

J.KATOEN; BAIER, C. **Principles of Model Checking**. [S.l.]: Massachusetts Institute of Technology, 2008.

LISBOA, E. et al. A Design Flow Based on a Domain Specific Language to Concurrent Development of Device Drivers and Device Controller Simulation Models. In: INTERNATIONAL WORKSHOP ON SOFTWARE AND COMPILERS FOR EMBEDDED SYSTEMS, 12., New York, NY, USA. **Proceedings...** ACM, 2009. p.53–60. (SCOPES '09).

MACIEIRA, R.; BARROS, E.; ASCENDINA, C. Towards more reliable embedded systems through a mechanism for monitoring driver devices communication. In: QUALITY ELECTRONIC DESIGN (ISQED), 2014 15TH INTERNATIONAL SYMPOSIUM ON. **Anais...** [S.l.: s.n.], 2014. p.420–427.

MACIEIRA, R.; LISBOA, E.; BARROS, E. Device Driver Generation and Checking Approach. In: COMPUTING SYSTEM ENGINEERING (SBESC), 2011 BRAZILIAN SYMPOSIUM ON. **Anais...** [S.l.: s.n.], 2011. p.72–77.

MCCARTHY, J. Towards a mathematical science of computation. **IFIP Congress**, [S.l.], p.21—-28, 1996.

MOORE, G. E. Cramming more components onto integrated circuits. **Electronics, Volume 38, Number 8**, [S.l.], 1965.

MOURA, L. de; BJORNER, N. Satisfiability Modulo Theories: an appetizer. **Springer-Verlag**, [S.l.], 2009.

MUKUND, M. Linear-Time Temporal Logic and Büchi Automata. **Tutorial talk,Winter School on Logic and Computer Science, Indian Statistical Institute,Calcutta**, [S.l.], 1997.

PNUELI, A. The temporal logic of programs. **In Proceedings of the 18th Annual Symposium on the Foundations of Computer Science**, [S.l.], 1977.

PNUELI, A.; MANNA, Z. **The temporal logic of reactive and concurrent systems**. New York: Springer-Verlag, 1992.

RESEARCH, M. Getting Started with Z3: a guid. **Microsoft Research**, [S.l.], 2016.

REVEILLERE, L. et al. A DSL approach to improve productivity and safety in device drivers development. **Proceedings ASE 2000. Fifteenth IEEE International Conference on Automated Software Engineering**, [S.l.], p.101–109, 2000.

ROZIER, K. Y. Linear Temporal Logic Symbolic Model Checking. **Computer Science Review**, [S.l.], 2010.

ROZIER, K. Y.; VARDI, M. Y. LTL Satisfiability Checking. **Springer-Verlag Berlin Heidelberg**, [S.l.], 2007.

SIMS, S.; BUTTS, K.; RANVILLE, S. Automated Validation of Software Models. In: ANNUAL INTERNATIONAL CONFERENCE ON AUTOMATED SOFTWARE ENGINEERING, 16. **Proceedings...** IEEE, 2002.

SISTLA, A.; VARDI, M.; WOLPER, P. Reasoning about infinite computation paths. **24th Annual Symposium on Foundations of Computer Science**, [S.l.], p.185–194, 1983.

SMULLYAN, R. **First-order Logic**. [S.l.]: Dover, 1995. (Dover books on advanced mathematics).

SWIFT, M. M. et al. Nooks: an architecture for reliable device drivers. In: WORKSHOP ON ACM SIGOPS EUROPEAN WORKSHOP, 10., New York, NY, USA. **Proceedings...** ACM, 2002. (EW 10).

TELCORDIA-BELLCORE. Feature requirements including: spcs capabilities and features. **Telcordia-Bellcore**, [S.l.], 1996.

TOYN, I.; GALLOWAY, A. Formal Validation of Hierarchical State Machines against Expectations. **Australian Software Engineering Conference**, [S.l.], p.101–109, 2007.

Z STANDARDS PANEL, D. by members of the. Formal Specification - Z Notation - Syntax Type and Semantics. **Consensus Working Draft 2.5**, [S.l.], 2000.