Pós-Graduação em Ciência da Computação

Rafael Melo Macieira

Técnica baseada em Contratos para a Validação da Comunicação de alto nível entre Software e Hardware

Universidade Federal de Pernambuco
posgraduacao@cin.ufpe.br
http://cin.ufpe.br/~posgraduacao

Recife
2018

Rafael Melo Macieira

Técnica baseada em Contratos para a Validação da Comunicação de alto nível entre Software e Hardware

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 Doutor em Ciência da Computação.

Orientador: Edna Natividade da Silva Barros

Recife
2018

Catalogação na fonte
Bibliotecária Monick Raquel Silvestre da S. Portes, CRB4-1217

M152t Macieira, Rafael Melo
Técnica baseada em contratos para a validação da comunicação de alto nível entre software e hardware / Rafael Melo Macieira. – 2018.
215 f.: il., fig., tab.

Orientadora: Edna Natividade da Silva Barros.
Tese (Doutorado) – Universidade Federal de Pernambuco. CIn, Ciência da Computação, Recife, 2018.
Inclui referências e apêndices.

1. Engenharia da computação. 2. Sistemas embarcados. I. Barros, Edna Natividade da Silva (orientadora). II. Título.

621.39 CDD (23. ed.) UFPE- MEI 2018-089

Rafael Melo Macieira

Técnica baseada em Contratos para a Validação da Comunicação de Alto Nível entre Software e Hardware

Tese de Doutorado apresentada ao Programa de Pós-Graduação em Ciência da Computação da Universidade Federal de Pernambuco, como requisito parcial para a obtenção do título de Doutor em Ciência da Computação.

Aprovado em: 05/03/2018.

_____

Orientadora: Profa. Dra. Edna Natividade da Silva Barros

BANCA EXAMINADORA

_____

Prof. Dr. André Luis de Medeiros Santos
Centro de Informática /UFPE

_____

Prof. Dr. Alexandre Cabral Mota
Centro de Informática /UFPE

_____

Prof. Dr. Victor Wanderley Costa de Medeiros
Departamento de Estatística e Informática / UFRPE

_____

Prof. Dr. Djones Vinicius Lettnin
Departamento de Engenharia Eletrônica / UFSC

_____

Prof. Dr. Fabian Luis Vargas
Departamento de Engenharia Elétrica / PUC-RS

Eu dedico esta tese à minha família, aos meus amigos e professores que deram apoio incondicional, força, incentivo e amizade sem igual. Sem eles nada disto seria possível.

AGRADECIMENTOS

Dedico e agradeço este momento e este trabalho à toda minha família, começando pelos meus pais, os quais sempre me deram total apoio, incentivo, força e amor para que este trabalho foi concluído. Chegamos lá, pai e mãe! Agradeço aos meus irmãos, que assim como meus pais, sempre me disseram palavras de apoio e sempre acreditaram em mim. Agradeço à minha esposa/companheira/parceira/amiga que com muito amor, muita paciência e compreensão, me apoiou incondicionalmente na realização deste trabalho e na realização deste sonho. Sem seu apoio no meu dia-a-dia não sei se teria conseguido. Amo muito vocês todos! Vocês fizeram este trabalho junto comigo.

Agradeço os meus professores da UFS, aos professores da UFPE e aos meus colegas tanto da graduação quanto da pós-graduação. Muitas noites perdidas de trabalho valeram à pena. Obrigado por todo apoio e ensinamentos. Agradeço à minha orientadora, Profª Edna Barros, que ao longo de 9 anos e com muita paciência e dedicação vem me ensinando valores técnicos e humanos. Só tenho a agradecer.

Agradeço tanto aos meus amigos de Aracaju quanto aos amigos que fiz em Recife por toda diversão e apoio em momentos de tristeza, cansaço e desânimo. Vocês também me deram forças para continuar. Obrigado!

Agradeço aos meus colegas e amigos de trabalho do Instituto SENAI de Inovação para TICs que me deram total suporte no final dessa jornada e que compreenderam o significado deste trabalho para minha vida, "segurando as pontas" sempre que foi preciso. Sou eternamente grato.

Obrigado a todos vocês por acreditarem em mim!

"Os créditos pertencem ao homem que está na arena, com a face coberta de poeira, suor e sangue; que luta com bravura, erra e, seguidamente, tenta atingir o alvo. É aquele que conhece os grandes entusiasmos, as grandes devoções e se consome numa causa justa. É aquele que, no sucesso, melhor conhece o triunfo final dos grandes feitos e que, se fracassa, pelo menos falha ousadamente, de modo que o seu lugar jamais será entre as almas tímidas, que não conhecem nem a vitória, nem a derrota"

(Theodore Roosevelt)

RESUMO

O uso de sistemas eletrônicos embarcados em um amplo espectro de aplicações aumentou substancialmente. Telefones celulares, computadores de bordo, dispositivos IoT e sistemas ciber-físicos são alguns exemplos em que sistemas embarcados exigem mais flexibilidade para o processamento de diferentes tipos de aplicações e protocolos de comunicação. A necessidade desta flexibilidade elevada requer o uso de processadores de propósito geral como uma solução para configurar e controlar uma quantidade crescente de periféricos. No entanto, isso também implica uma necessidade crescente do uso de software dependente de hardware (HdS). Uma vez que HdS é um componente altamente crítico e propenso a erros devido à natureza do ambiente no qual está inserido e a sua difícil codificação, é importante que as fases de desenvolvimento e de operação destes componentes sejam suportadas por metodologias que tornem mais explícitas violações de acessos a dispositivos através do acompanhamento de de garantias e suposições especificadas em contratos definidos com base na especificação dos protocolos de comunicação de alto nível entre processadores e estes dispositivos. Assim, esta abordagem propõe uma técnica *Built-in Self Test*(BIST) fundamentada na formalização destes contratos e da validação em tempo de execução de propriedades temporais de protocolos de comunicação de alto nível entre os dispositivos e seus *device drivers*, as quais compões tais contratos e são comumente especificadas com base em uma documentação informal (*datasheets*), durante a simulação de plataformas virtuais ou execução de plataformas de hardware real. Para tornar essa técnica viável juntamente com metodologias consolidadas, uma linguagem específica foi desenvolvida para suportar a sua utilização como uma melhoria de modelos baseados na plataforma, permitindo um refinamento incremental do protocolo de comunicação e das especificações das propriedades temporais juntamente com o refinamento da plataforma. Alguns experimentos usando um dispositivo DM9000A Ethernet mostraram que esta abordagem é eficaz na detecção de erros que provocam falhas críticas indesejada em HdS, consumindo pouco tempo de projeto e impactando pouco ou, para plataformas de hardware real, sem impacto algum no tempo de execução.

Palavras-chaves: Device driver. Validação. Sistemas embarcados. Monitoramento. Projeto de sistemas. Projeto para testabilidade.

ABSTRACT

The use of electronic embedded system for general or multi-purpose applications has increased substantially. Mobile phones, vehicles computers and IoT and cyber-physical systems are some examples in which embedded systems require more flexibility for processing different types of applications and communication protocols. The need for this high flexibility requires the use of general purpose processors as a solution for configuring and controlling a considerable amount of peripherals. However, this also implies an increasing need for hardware-dependent software (HdS). Since HdS is a highly critical component and error prone due to the nature of the environment in which it is inserted and its hard coding, it is important that its development and runtime phases are supported by methodologies that make more explicit devices' accesses violations, through the monitoring of contracts defined based on the communication protocol specification. Thus, this approach proposes a Built-in Self Test(BIST) technique base on the formalization and runtime validation of temporal properties in high-level communication protocols between devices and drivers, commonly specified in informal documentation (datasheets), during the simulation of virtual platforms or execution of hardware platforms. To make this technique feasible together with consolidated methodologies, a domain-specific language was designed to support its use as an improvement of platform-based designs, allowing an iterative refinement of the communication protocol and temporal properties specifications along with the refinement of the platform. Some experiments using an DM9000A Ethernet device and an Altera UART showed that this approach is effective in detecting errors that provoke undesired critical failures in HdS, consuming little design time and little or, for real hardware platforms, no execution time increasing.

Key-words: Device driver. Validation. Embedded systems. Monitoring. System design. Design for testability.

LISTA DE ILUSTRAÇÕES

LISTA DE TABELAS

LISTA DE SIGLAS

3G	THIRD GENERATION
4G	FOURTH GENERATION
ASIC	APPLICATION SPECIFIC INTEGRATED CIRCUIT
ASIP	APPLICATION-SPECIFIC INSTRUCTION SET PROCESSOR
BA	BÜCHI AUTOMATON
BCAM	BUS CYCLE-ACCURATE MODEL
BCL	BLOCK-BASED CONTRACT LANGUAGE
BDD	BEHAVIOR-DRIVEN DEVELOPMENT
BIST	BUILT-IN SELF TEST
BSI	BUS SLAVE INTERFACE
BSPI	BUS SNOOPING INTERFACE
CAM	CYCLE-ACCURATE MODEL
CBD	COMPONENT-BASED DESIGN
CFG	CONTROL-FLOW GRAPH
COTS	COMMERCIAL OFF-THE-SHELF
CPS	CYBER-PHYSICAL SYSTEM
CPU	CENTRAL PROCESSING UNIT
DOS	DENIAL OF SERVICE
DS	DEVICE SELECTOR
DSL	DOMAIN-SPECIFIC LANGUAGE
DSP	DIGITAL SIGNAL PROCESSOR
ECA	EVENT CLOCK AUTOMATA
ERE	EXTENDED REGULAR EXPRESSION
ETU	EMBEDDED TRACE UNIT
FLTL	FINITE LINEAR TEMPORAL LOGIC
FPGA	FIELD PROGRAMMABLE GATE ARRAY

GPS	GLOBAL POSITIONING SYSTEM
GPU	GRAPHICS PROCESSING UNIT
HAL	HARDWARE ABSTRACTION LAYER
HDL	HARDWARE DESCRIPTION LANGUAGE
HDS	HARDWARE-DEPENDENT SOFTWARE
ICMP	INTERNET CONTROL MESSAGE PROTOCOL
IEEE	INSTITUTE OF ELECTRICAL AND ELECTRONICS ENGINEERS
IOT	INTERNET OF THINGS
IRQ	INTERRUPT REQUEST
ISO	INTERNATIONAL ORGANIZATION FOR STANDARDIZATION
ISS	INSTRUCTION SET SIMULATOR
LDC	LINHAS DE CÓDIGO
LED	LIGHT EMITTING DIODE
LTL	LINEAR TEMPORAL LOGIC
MBD	MODEL-BASED DESIGN
MDDC	MONITOR OF DRIVER / DEVICE COMMUNICATION
MOC	MODEL OF COMPUTATION
MOP	MONITOR ORIENTED PROGRAMMING
MPSOC	MULTIPROCESSOR SYSTEM-ON-CHIP
NFC	NEAR FIELD COMMUNICATION
NOC	NETWORK-ON-CHIP
OEM	ORIGINAL EQUIPMENT MANUFACTURER
OSI	OPEN SYSTEMS INTERCONNECTION
PBD	PLATFORM-BASED DESIGN
PCI	PERIPHERAL COMPONENT INTERCONNECT
PSL	PROPERTY SPECIFICATION LANGUAGE
PT	PROTOCOL TRANSLATOR
RTC	REAL TIME CLOCK
RTL	REGISTER TRANSFER LEVEL
RTOS	REAL-TIME OPERATING SYSTEM

RVU	RUNTIME VERIFICATION UNIT
SCTC	SYSTEMC TEMPORAL CHECKER
SD	SECURE DIGITAL
SER	SPECIFY-EXPLORE-REFINE
SIM	SUBSCRIBER IDENTITY MODULE
SOC	SYSTEM-ON-CHIP
STL	SIGNAL TEMPORAL LOGIC
SUT	SYSTEM-UNDER-TEST
TDD	TEST-DRIVEN DEVELOPMENT
TDEVC	TEMPORAL DEVC
TLM	TRANSACTION-LEVEL MODELING
USB	UNIVERSAL SERIAL BUS
WPG	WAYPOINT GRAPH

SUMÁRIO

1 INTRODUÇÃO

O uso de sistemas eletrônicos embarcados para aplicações de múltiplos propósitos aumentou substancialmente nos últimos anos. Telefones celulares, computadores de bordo, sistemas ciber-físicos (Cyber-Physical System (CPS)) e *internet das coisas* (Internet of Things (IoT)) são alguns exemplos em que os sistemas embarcados exigem mais flexibilidade para processar diferentes tipos de aplicações e protocolos de comunicação e controle.

Com a redução do tamanho dos circuitos integrados, sua crescente capacidade de integração e baixo custo, torna-se economicamente e tecnologicamente viável o uso de processadores de propósito geral em sistemas embarcados, agregando mais poder de processamento a estes sistemas. Para se ter uma ideia, 90% dos processadores produzidos no mundo vem sendo aplicados em sistemas embarcados (LETTNIN; WINTERHOLER, 2017).

Este foi um fator fundamental para que houvesse o aumento na flexibilidade e diversidade de aplicações em dispositivos cada vez menores. A alta capacidade de integração permitiu também que não só mais unidades de processamento de propósito geral fossem utilizadas, mas também a integração de coprocessadores e controladores de periféricos em uma única pastilha de silício, unificando sistemas que anteriormente necessitavam de grandes áreas em apenas um único chip, conhecidos como *Systems-on-chip* System-on-Chip (SoC).

SoC, sejam eles com uma única unidade de processamento ou com múltiplos processadores (Multiprocessor System-on-Chip (MPSoC)), em muitos casos contêm dezenas de controladores de periféricos. Para se ter uma ideia, versões mais simples de telefones móveis convencionais atualmente contêm, por exemplo, módulos de áudio, Universal Serial Bus (USB), Near Field Communication (NFC), Global Positioning System (GPS), bateria, cartão Subscriber Identity Module (SIM), cartão Secure Digital (SD), memória flash, Wi-Fi Institute of Electrical and Electronics Engineers (IEEE) 802.11, Bluetooth e módulo de telefonia (Third Generation (3G)/Fourth Generation (4G)), câmeras de vídeos, Graphics Processing Unit (GPU), *touchscreen*, acelerômetros, *vibracall*, Light Emitting Diode (LED) etc.

A flexibilidade agregada nestes sistemas provém justamente do reuso, configuração e do compartilhamento destes recursos de hardware. Logo, se faz necessário seu controle por parte do software em execução na(s) unidade(s) de processamento. Em muitos casos, projetistas fazem uso de componentes padronizados e pré-fabricados, conhecidos como *components off-the-shelf* (Commercial Off-The-Shelf (COTS)), para minimizar os custos de projetos devidos às suas produções em escala. Nestes casos se fazem necessárias configurações prévias via software com o intuito de adaptação aos requisitos do sistema e do ambiente, ou configurações posteriores quando o produto já está no mercado, também via software, buscando ajustes ou adaptações de manutenção (ECKER; MUELLER; DOMER,

2009).

Desta forma, a pilha de software em sistemas embarcados atualmente já deixou de ser contabilizada em dezenas ou centenas de bytes. Para se ter uma ideia, de acordo com Pretschner et al. (2007), em 2010, software aplicado para a área automotiva alcançava por volta de 1 GB. Isso se dá pelo fato de que o software embarcado é aplicado em diversas camadas no sistema, desde aplicações em alto nível até controle de componentes de baixo nível (LETTNIN; WINTERHOLER, 2017).

Para que o software embarcado que controla todo o sistema também tenha controle de uso e configuração dos recursos de hardware, é preciso que haja partes na camada de software especializadas em conhecer e se comunicar com os controladores de hardware destes periféricos, provendo assim uma interface para um acesso direto e simplificado às características e funcionalidades destes recursos.

Estas partes de software, conhecidas como *hardware-dependent software (HdS)*, em português, software dependente de hardware, são fortemente acopladas e específicas para blocos de hardware distintos. Desta maneira, a existência de um HdS não faz sentido sem o seu hardware, uma vez que estes dois juntos implementam funcionalidades do sistema (ECKER; MUELLER; DOMER, 2009).

Como dito anteriormente, o número de periféricos em um sistema vem aumentando com o avanço da tecnologia, e o aumento no número de periféricos implica em um aumento do HdS. A Figura 1 mostra a relação entre o esforço gasto em um projeto de SoC e o esforço gasto para desenvolver HdS, em função da minimização do tamanho dos transistores em SoCs.

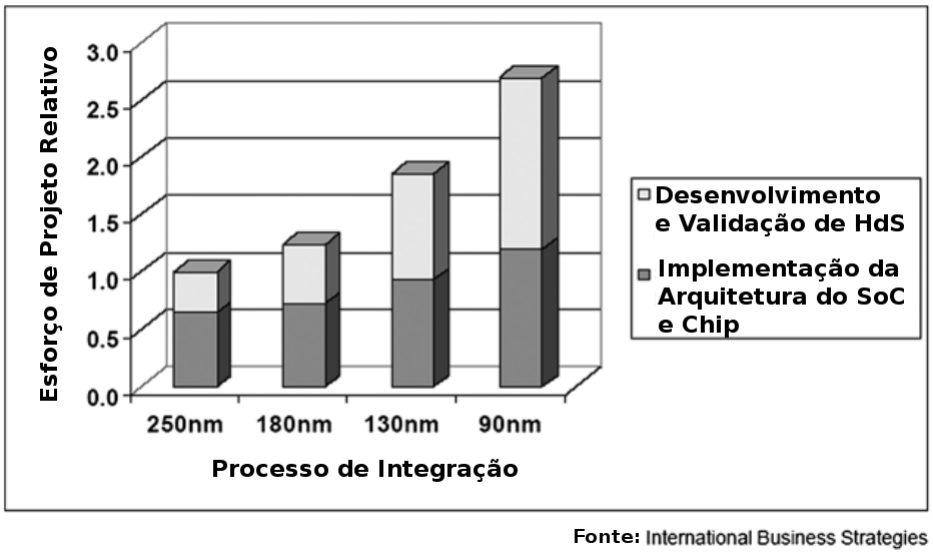


Figura 1 – Relação entre o esforço gasto em projetos de SoC e projetos de HdS. Fonte Traduzida: (ECKER; MUELLER; DOMER, 2009)

A Figura 1 confirma exatamente que com a diminuição do tamanho do transistor é possível conceber componentes de hardware mais complexos e com quantidades maiores

de dispositivos integrados. Com isso exige-se mais software dependente dos controladores destes dispositivos.

Incluídos nesta categoria de software estão os *device drivers*. Os *device drivers* são tipos de HdS que provêm acesso do software aos recursos do hardware (ECKER; MUELLER; DOMER, 2009). Sendo assim, eles compõem uma camada de software responsável pelo interfaceamento entre componentes de software e componentes de hardware, interpretando e traduzindo todas as requisições realizadas pelas mais altas camadas de software em um protocolo de comunicação específico do dispositivo de hardware para o qual o *driver* foi projetado.

O desenvolvimento e teste de *drivers* consomem muito tempo e são extremamente complexos, e por isso são bastante propensos a erros (REVEILLERE et al., 2000). Por exemplo, testar se um *device driver* está tratando corretamente uma falha do dispositivo, implica em simular este componente exatamente durante a ocorrência de falha. Outro problema que é enfrentado por desenvolvedores de *drivers* é o fato de que estes componentes podem suportar controladores de vários dispositivos e, em muitos casos, o desenvolvedor não tem acesso a nenhum deles. Estas dificuldades podem implicar em situações críticas em que nem sequer os *drivers* sejam validados funcionalmente. Muitos *device drivers* implementados para o sistema operacional Linux contêm comentários do tipo: "apenas teste de compilação." (RENZELMANN; KADAV; SWIFT, 2012), indicando que as únicas análises pelas quais o HdS passou foram a análise sintática e semântica da linguagem na qual ele foi implementado.

No domínio dos sistemas embarcados o cenário ainda é um pouco mais problemático. Sistemas embarcados são projetados para plataformas de hardware de aplicação específica e cada sistema requisita o projeto de *drivers* específicos ou o porte de alguns *drivers* já desenvolvidos (REVEILLERE et al., 2000). Por isso, em muitos casos *drivers* são desenvolvidos a partir da cópia e edição de *templates* de códigos de *drivers* já existentes, normalmente sem um entendimento mais aprofundado, o que implica em erros que seriam facilmente evitados se estes *drivers* fossem codificados do início, sem o uso de um código base (SWIFT et al., 2002).

Um estudo sobre erros e confiabilidade de software, realizado por Ostrand e Weyuker (2002), mostra que códigos de software contêm de 2 a 75 erros a cada 1000 linhas. Considerando um cenário otimista de apenas 2 erros a cada 1000 linhas e levando em consideração que o *Linux Kernel* versão 4.5, lançado em 2016, tem aproximadamente 17 milhões de linhas de código (LINUX..., 2016), estima-se a existência de aproximadamente 34.000 erros. Levando em consideração que aproximadamente 70% do *Linux Kernel* é composto por *device drivers*, e estes ainda têm uma taxa de erro de aproximadamente 3 a 7 vezes maior do que qualquer outro código do *Kernel*, o cenário é ainda mais crítico (GANAPATHI; GANAPATHI; PATTERSON, 2006; TANENBAUM; HERDER; BOS, 2006; SWIFT et al., 2002; CHOU et al., 2001).

Para *device drivers*, além da alta propensão à inserção de erros durante sua codificação,

a sua execução é extremamente crítica. Como são componentes do *Kernel* e sua execução acontece em modo supervisor, é permitido acesso a qualquer recurso e em qualquer endereçamento do hardware. Desta forma, erros nestes componentes podem gerar falhas graves e comprometer todo o sistema.

Quando sistemas ciber-físicos (CPS) são levados em consideração, todo o projeto se torna ainda mais complexo e sua execução ainda muito mais crítica. Sistemas ciber-físicos integram computação com processos físicos. Eles são compostos por computadores embarcados, monitores de rede e controladores de processos físicos, geralmente com *feedback loops*, onde processos físicos, através de sensores e atuadores, interferem e orientam a computação e vice-versa (LEE, 2008). É muito comum vê-los aplicados a ambiente críticos, tais como transporte massivo de pessoas, sistemas de segurança, usinas de geração de energia e equipamentos médicos, onde várias vidas e estruturas físicas de alto custo são envolvidas. Assim, mesmo sendo um grande desafio projetar tais sistemas, a existência de falhas em sua execução é inaceitável (CLARKE; GRUMBERG; PELED, 1999). A professora Marilyn Wolf, do *Georgia Institute of Technology*, deixa claro o desafio de se projetar CPS:

> *"Errors of any kind – component failures, design flaws, etc. – can injure or kill people. Not only must these systems be carefully verified, but they must also be architected to guarantee certain properties."* (WOLF, 2014)

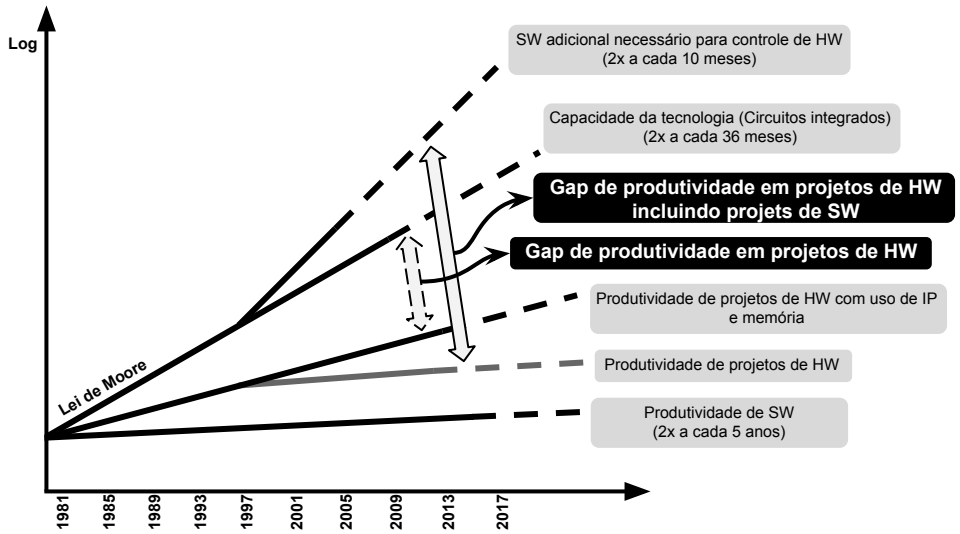


Figura 2 – Gap de produtividade. Redesenhado a partir de (ITRS..., 2015)

Fica claro que se faz necessário conceber sistemas com um alto grau de dependabilidade. De acordo com Knight (2012), dependabilidade de um sistema é a sua habilidade de evitar a ocorrência de falhas de serviços em frequências e severidades maiores do que o aceitável.

Porém, apesar de toda a criticidade destes componentes e a necessidade de implementá-los de maneira correta, ainda existe uma grande defasagem de produtividade no processo de desenvolvimento de sistemas computacionais. Como pode ser visto na Figura 2 e

fundamentado na Lei de Moore (MOORE, 1965), o número de transistores que podem ser integrados num único chip dobra a cada 36 meses. No entanto, devido à complexidade atual dos dispositivos, a produtividade no desenvolvimento de projetos de hardware não consegue acompanhar este progresso

Para tornar o cenário ainda pior, a produtividade de projetos de software também não consegue acompanhar nem mesmo a produtividade dos projetos de hardware, crescendo apenas 2 vezes a cada 5 anos. Isto vai de encontro à necessidade do desenvolvimento de software dependente de hardware (HdS), que cresce 2 vezes a cada 10 meses. Assim, existe a necessidade de otimizar o desenvolvimento de camadas de software, reduzindo o tempo e esforço de projeto exigidos e, principalmente, mantendo ou aumentando a confiabilidade do sistema.

Atualmente diversas técnicas e metodologias de projetos são aplicadas no desenvolvimento de sistemas embarcados. Projetos baseados em componentes (Component-based Design (CbD)), baseados em modelos (Model-based Design (MbD)) e baseados em plataforma (Platform-based Design (PbD)) são algumas metodologias que tentam viabilizar o desenvolvimento de sistemas apostando em modelos de mais alto nível de abstração de componentes que são integrados compondo plataformas virtuais, onde estas são refinadas até se chegar a uma plataforma alvo (sistema hardware e software). A partir destas metodologias, diversos outros trabalhos surgiram com o intuito de melhorá-las, uma vez que vários problemas pontuais ainda estão longe de serem resolvidos.

É importante deixar claro que o objetivo destas técnicas é mapear de maneira incremental requisitos de uma determinada aplicação, normalmente especificados em documentos descritos em uma linguagem natural, em um sistema composto por componentes de computação e comunicação mapeados em uma plataforma de hardware controlada por camadas de software.

Contudo, apesar da complexidade, este processo deve resultar em um sistema dependável, onde os seus serviços sejam íntegros, estejam disponíveis, executem corretamente em componentes de computação que se comunicam corretamente através de componentes de comunicação igualmente dependáveis. Este processo é fundamental, uma vez que os custos de reparação de um defeito em um sistema durante sua fase de manutenção e operação são 500 vezes mais elevados comparados a uma correção em uma fase inicial do projeto, sem contar com a consequente baixa na reputação tanto do sistema quanto do seu fabricante (BAIER; KATOEN, 2008).

Um dos grandes desafios se encontra justamente na comunicação entre processadores e periféricos em uma plataforma, como dispositivos de entrada e saída, dispositivos de armazenamento de dados e co-processadores ou aceleradores em hardware. Esta comunicação se torna mais difícil e propensa a erros principalmente quando envolve o acoplamento de componentes desenvolvidos por terceiros, por fabricantes do equipamento original (Original Equipment Manufacturer (OEM)) ou COTS. As interfaces e seus requisitos de uso, como

os protocolos de comunicação, por exemplo, normalmente são descritos em linguagens naturais e informais. Tais informações essenciais escritas de maneira confusa e que gerem diferentes interpretações por parte de equipes distintas de projeto pode resultar em um sistema não confiável (SANGIOVANNI-VINCENTELLI; DAMM; PASSERONE, 2012). Assim, descrever modelos bem estruturados de interfaces de componentes e protocolos é um desafio atual no projeto de sistemas.

Partindo do que foi mencionado até o momento, é possível perceber que há uma lacuna nas metodologias atuais de desenvolvimento de software dependente de hardware, especialmente os responsáveis pelo protocolo de comunicação entre o software executando em processadores e os controladores de periféricos, como dispositivos de entrada e saída, de armazenamento e módulos de hardware de aplicação única.

Esta lacuna se encontra principalmente na especificação informal dos protocolos de comunicação de alto nível e no fato de que essas metodologias não acompanham de maneira incremental a complexa, crítica, lenta e custosa implementação destes protocolos especificamente. Na ocorrência de uma má interpretação de protocolos ao longo do desenvolvimento do sistema, o controle de periféricos pode ser totalmente comprometido, assim como o comportamento do software pode ser imprevisível em uma situação de uso indevido do periférico, o pode ser catastrófico.

1.1 Motivação

Apesar dos problemas relatados anteriormente, algumas metodologias tentam resolver os problemas relacionados ao desenvolvimento de *device drivers*. Diversos estudos propõem o uso de modelos em alto nível para síntese destes componentes, apostando na sua corretude por construção. Entretanto, estas técnicas pecam pelo fato de que, quando o modelo é simplificado, somente partes destes componentes são construídas, exigindo ainda um grande esforço de codificação manual. Porém, quando os modelos são bem complexos, suas descrições são muito semelhantes às linguagens de propósito geral, indo de encontro à simplicidade de modelos de alto nível propostas pela metodologia baseadas em modelos (MbD).

Outras metodologias tentam atacar o problema através da especificação e verificação de restrições ou propriedades que o software deve respeitar. Estas metodologias se baseiam no conceito de contratos, nos quais componentes integrados em um ambiente devem respeitar as cláusulas estipuladas para que haja o correto uso das suas interfaces. Estes contratos são formalizações da correta integração de componentes (contratos horizontais), além de garantir que modelos em níveis mais baixos de abstração estejam consistentes com os modelos nos níveis mais altos e que os modelos de componentes sejam representações confiáveis dos componentes disponíveis no projeto (contratos verticais) (SANGIOVANNI-VINCENTELLI; DAMM; PASSERONE, 2012).

Assim, em projetos baseados em contratos, modelos de componentes tem associados contratos às suas interfaces. As cláusulas destes contratos são asserções representando **suposições** por parte do ambiente e **garantias** por parte do componente. Em outras palavras, o componente, **supondo** que o ambiente esteja fazendo seu uso corretamente do acordo com o contrato, deve então **garantir** as suas ações estipuladas nas cláusulas em relação ao meio.

Atualmente, algumas abordagens propõem garantir a corretude da implementação de *device drivers* se baseando na especificação de propriedades formais que representam requisitos ou propriedades que estes devem respeitar. Entretanto, normalmente tais abordagens ou o fazem utilizando modelos em nível de sinais de entrada e saída para representar suas interfaces ou requisitos de uso, ou então fazem uso de asserções globais e genéricas para todo software embarcado.

Para o primeiro caso, o problema se encontra no fato de que especificar o protocolo de comunicação de alto nível entre um componente de software e um dispositivo com base em modelos em nível de sinais lógicos, torna-se necessário expressar detalhes que não são importantes ou não dizem respeito à configuração e ao uso deste dispositivo. É interessante abstrair certos detalhes como por exemplo protocolo de barramento, traduções de endereços, rotas por diferentes barramentos e etc. Vale ressaltar que protocolos de comunicação de alto nível entre *drivers* e dispositivos são independentes do meio de transmissão dos dados. Mesmo variando os tipos de barramentos, adicionando roteadores, pontes ou transdutores, o protocolo de comunicação definido na especificação do dispositivo e responsável pelo uso e configuração do mesmo permanece inalterado. Assim, é necessário ter modelos especializados destes protocolos implementados pelos *device drivers* que sejam agregados a estes modelos genéricos (*motivação I*).

O segundo caso acaba sendo semelhante ao primeiro, porém seu agravante está em não acompanhar o refinamento ou incremento dos modelos na plataforma. Refinar o modelo significa especializar ou adicionar detalhes a componentes previamente especificados nos modelos. Em resumo, significa um salto para níveis de abstração mais baixos. Incrementar o modelo significa adicionar componentes que até o momento não existiam e sem necessariamente mudar o nível de abstração.

Como cada refinamento ou incremento significa incluir mais detalhes nos modelos (informação de tempo, limite de dados, etc.). Quando se tem asserções globais, a cada refinamento ou incremento estas asserções devem ser modificadas, substituídas ou adaptadas para os novos detalhes incluídos. Isto pode resultar em erros na redefinição destas propriedades ou na criação de expressões muito grandes e complexas devido à agregação de propriedades mais detalhadas em uma única expressão. Logo, existe a necessidade da especificação de modelos especializados que suporte refinamentos e incrementos dos modelos dos componentes através da agregação de propriedades (asserções) de maneira hierárquica, onde, à medida que o modelo é refinado ou incrementado, as asserções nos altos níveis

de abstrações sejam agregadas às asserções nos níveis mais baixos de abstração. Desta maneira, as asserções englobam todo o modelo, porém são descritas incrementalmente através de asserções distintas e de escopos locais. (*motivação II*).

Ficou claro que projetar *device drivers* confiáveis é um dos grandes desafios do estado-da-arte envolvendo sistemas embarcados. Entretanto, um outro grande desafio é mantê-los confiáveis, seguros e íntegros durante a sua execução. É muito comum dispositivos sofrerem atualização de software durante a sua fase de operação e em muitas dessas atualizações estão incluídos códigos (principalmente *device drivers*) de terceiros que podem não ter passado por processos de testes ou verificações adequados.

Além de atualizações de software, a alta conectividade de dispositivos (dispositivos IoT, por exemplo) é uma das grandes preocupações de especialistas em segurança (*security*) de sistemas embarcados. Intrusões e ataques podem acarretar em negação de serviços (Denial of Service (DoS)) e alteração no comportamento do sistema, o que pode resultar em situações desastrosas quando relacionado a CPS. Um *device driver* pode ser visto como uma porta de entrada para ataques e uma inserção de código pode facilmente interromper ou alterar seu comportamento.

É interessante que o sistema esteja pronto para detectar e reagir a situações críticas, sendo tolerante a falhas. Várias pesquisas (SWIFT et al., 2002; GANAPATHY et al., 2007; HERDER et al., 2007; WEGGERLE et al., 2011; KADAV; RENZELMANN; SWIFT, 2009) mostram a importância e a viabilidade dos sistemas tolerante a falhas, os quais isolam as falhas, tomam o controle sobre o sistema, levando-o para um estado seguro. Para tolerar falhas, o sistema deve estar pronto para identificar ou prever estes comportamentos críticos.

Uma maneira de tornar isso viável é fazer uso dos mesmos requisitos especificados durante a fase de desenvolvimento também durante a fase de operação do dispositivo para monitorar a integridade dos seus serviços. A vantagem disto é que os modelos especificados de maneira incremental e refinados durante a fase de desenvolvimento do dispositivo tiveram a sua corretude validada durante o processo. Assim, mesmo que qualquer alteração seja feita na camada de software, estes requisitos validados deverão ser respeitados em qualquer momento da execução (*motivação III).* Este é o principal motivo de esta técnica ser aplicada em tempo de execução e de maneira *online*.

1.2 Objetivo

Com base nas motivações apresentadas, este trabalho tem como objetivo principal desenvolver uma técnica que auxilie o desenvolvimento (*motivações I e II*) de *device drivers* **confiáveis** e **seguros** através da identificação de violações de propriedades dos protocolos de comunicação entre componentes de uma plataforma embarcada, durante a sua simulação e que possa ser aplicada também na garantia da **integridade** do sistema durante a sua execução (*motivação III*), em fase de operação.

1.3 Abordagem Proposta

Com um objetivo bem definido para atacar a problemática detalhada no início deste capítulo, este trabalho tem como abordagem fundamental a observação e validação da comunicação realizada entre processadores e periféricos de plataformas embarcadas com base na técnica Built-in Self Test (BIST).

Como explicado anteriormente e apresentado na Figura 3, em uma plataforma embarcada, o software em execução em um processador se comunica com seus periféricos através de elementos de comunicação como, por exemplo, barramentos, roteadores e transdutores. Cada periférico é logicamente integrado nesta plataforma através de um endereçamento, ou seja, cada periférico está enquadrado em uma faixa de endereços de memória que podem ser acessados pelo processador.

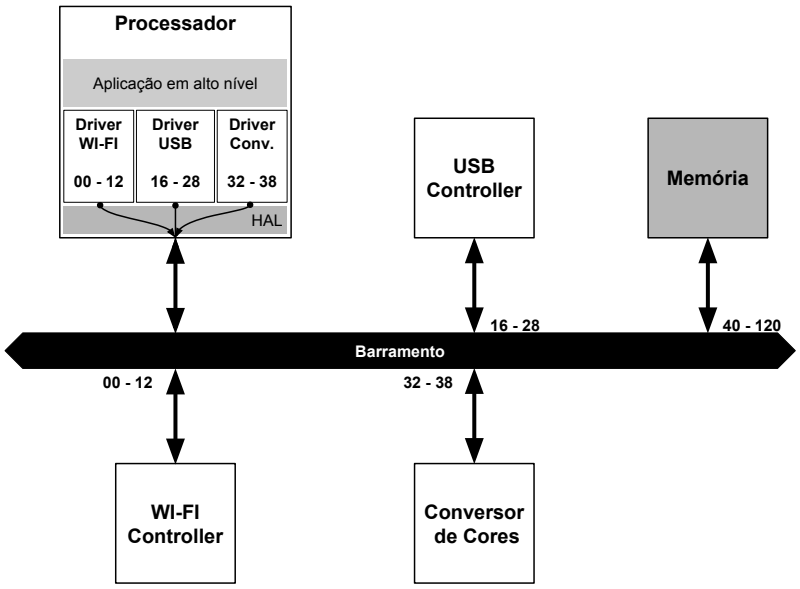


Figura 3 – Exemplo de uma plataforma embarcada.

A plataforma apresentada na Figura 3 contém basicamente 5 componentes interligados através de um barramento. Um processador, uma memória, um controlador WI-FI, um controlador USB, e um acelerador em hardware que executa a tarefa de converter imagens para diferentes espaços de cores. Com exceção do processador, todos os outros elementos estão mapeados na plataforma através de endereços de memória, como também pode ser visto na Figura 3.

Quando o software necessita de algum serviço oferecido por algum dos periféricos, as altas camadas da pilha de software solicitam ao *device driver* especifico daquele dispositivo a realização deste serviços.

O *driver* então traduz esta solicitação em uma sequência de escritas e leituras nos endereços contidos na faixa de endereçamento do dispositivo em questão. Esta sequência de escritas e leituras nos registradores dos dispositivos através dos endereços mapeados

é o que chamamos de protocolos de comunicação de alto nível entre os componentes da plataforma.

Assim, esta abordagem propõe um mecanismo que observa esta sequência de acessos aos endereços e, baseando-se em um modelo de referência, identifica se o protocolo de comunicação de alto nível está sendo respeitado com base na especificação de uso do dispositivo.

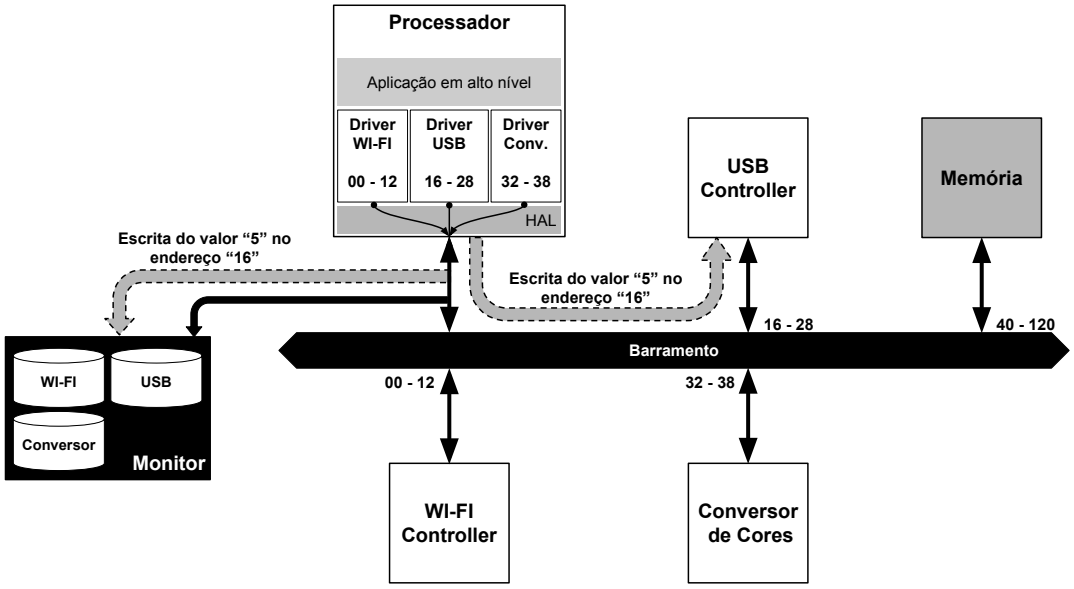


Figura 4 – Exemplo de uma plataforma embarcada com a técnica proposta aplicada.

A Figura 4 mostra que, para que esta observação seja realizada, é proposto também um monitor integrado à plataforma que intercepte todas as requisições realizadas pelo processador ao barramento da plataforma. A Figura 4 mostra que, quando o software em execução no processador requisita uma escrita no endereço 16, referente ao dispositivo *USB Controller*, o monitor também fica ciente desta requisição.

Contando com um modelo de referência internamente, o monitor verifica se aquele acesso, neste caso uma escrita de um valor (5) em um determinado endereço (16), é um comportamento esperado pelo dispositivo naquele momento e no estado de execução no qual ele se encontra.

Porém, a construção deste monitor é uma tarefa tão complexa quanto a implementação correta do *device driver*. Por esse motivo, esta abordagem também propõe a síntese deste monitor com base em uma especificação em alto nível do modelo de referência do protocolo dos dispositivos.

Para isto, foi desenvolvida uma linguagem de domínio específico, conhecida como Temporal DevC (TDevC), que permite, de maneira incremental, a especificação dos protocolos de comunicação em alto nível, bem como a especificação de suas restrições de usos. É importante deixar claro que no início deste trabalho cogitou-se a utilização do formalismo *Statechart* (HAREL, 1987) como *front-end* da abordagem. Porém, como se

desejava uma especificação tanto do protocolo de comunicação quanto de suas restrições de uso bastante específica para interfaceamento e comunicação entre componentes de hardware e software, optou-se por desenvolver e formalizar uma linguagem própria que agregasse todas as características fundamentais deste tipo de aplicação e que fosse fundamentada no formalismo *Statechart*.

Vale ressaltar que esta escolha nos permitiu ter total domínio sobre o modelo especificado do protocolo de comunicação e suas restrições de uso, possibilitante também a formalização e especificação de um formato intermediário para representar este modelo totalmente específico para a natureza do problema estudado.

Neste sentido então, esta abordagem propõe a síntese dos monitores a partir destas descrições em TDevC, tanto em modelos em SystemC, para plataformas virtuais em alto nível de abstração, como em módulos de hardware real, para efetuar de maneira não intrusiva na execução do software a validação das restrições dos protocolos de comunicação, tanto em simulação quanto em execução real.

Uma ferramenta chamada de *TDevCGen* foi desenvolvida para permitir tanto a especificação dos modelos *TDevC* quanto a síntese automática destes monitores.

Três experimentos foram realizados para validar a técnica. O primeiro experimento validou a sua viabilidade de uso, mostrando que a técnica reduz consideravelmente o esforço de implementação para se ter um ambiente de validação, e a sua eficácia, mostrando que as restrições especificadas foram identificadas.

Outro experimento diz respeito à medição do impacto na Execução do sistema com a técnica aplicada. Este experimento mostrou que não há impacto relacionado à execução do software, sendo o monitor totalmente transparente para o software e não intrusivo quanto ao seu tempo de execução e ao consumo de memória do sistema.

O último experimento realizou a comparação desta abordagem com outra técnica semelhante e de proposta parecida. Concluiu-se então que a técnica proposta nesta tese, por contar com uma especificação em alto nível do protocolo de comunicação de alto nível entre processadores e periféricos, reduz de maneira significativa o esforço de geração do ambiente de validação comparada com a outra técnica, além de suportar sua aplicação em simulações sem precisão de tempo, ou seja, simulações totalmente funcionais.

1.3.1 Principais Contribuições

Durante o cumprimento do objetivo principal desta abordagem, algumas **contribuições** se tornaram evidentes. São elas:

1. Com a necessidade de expressar as propriedades associadas aos protocolos de comunicação de alto nível entre elementos de processamento em uma plataforma, esta abordagem definiu formalmente a *TDevC*, uma linguagem de domínio específico (Domain-Specific Language (DSL)) para a descrição de protocolos de alto nível de

comunicação entre componentes de uma plataforma embarcada. Além do protocolo de comunicação, a *TDevC* também suporta a especificação de asserções que expressam restrições deste protocolo. São estas asserções que representam as garantias dos contratos especificados.

2. Para que esta abordagem suporte as principais metodologias de desenvolvimento de sistemas atuais e de definição e validação de propriedades formais, a *TDevC* foi concebida para dar suporte a hierarquia. Assim, é possível especificar protocolos de comunicação e suas propriedades de maneira incremental, acompanhando o desenvolvimento também incremental do sistema, com base em projetos baseados em modelos e plataforma. Isto permite que esta técnica seja integrada nas metodologias de projetos de sistemas atuais, possibilitando que a abordagem seja aplicada por empresas relevantes do setor no desenvolvimento de seus projetos.

3. Além de dar suporte à descrição dos protocolos de comunicação e suas propriedades, esta técnica provê a validação das especificações em TDevC para garantir que não haja inconsistências em relação ao protocolo descrito e nem contradições entre as propriedades. Nenhum dos trabalhos relacionados avaliados por esta tese provê a validação de seus modelos.

4. Para concretizar o objetivo principal deste trabalho é preciso que a comunicação entre os *device drivers* e seus respectivos dispositivos seja monitorada e que violações de restrições sejam identificadas. Assim, nesta tese desenvolvemos uma síntese automática de monitores da comunicação entre dispositivos de uma plataforma, a partir de descrições em *TDevC*.

5. Para que os monitores suportem à fase de desenvolvimento e a fase de operação do sistema, a síntese dos monitores resulta em modelos *SystemC*, aplicáveis a plataformas virtuais, e também em componentes de hardware, aplicáveis a plataformas reais.

6. Por fim, para que estes monitores não alterem características da execução do software embarcado, eles não devem depender de características de implementação ou de informações relacionadas à codificação das camadas de software. Dessa maneira, esta abordagem provê monitores que suportam integração na plataforma de maneira não intrusiva, preservando assim as características de desempenho, consumo de memória e consequentemente características temporais da execução do software.

1.4 Organização do Documento

A seguir, no Capítulo 2, este documento irá detalhar brevemente sobre protocolos de comunicação entre componentes de uma plataforma, *device drivers*, MbD, PbD e projetos baseados em contratos.

Em seguida, no Capítulo 3 serão apresentadas abordagens que também objetivam gerar e garantir a implementação de *device drivers* robustos.

O Capítulo 4 irá detalhar o fluxo geral da técnica proposta.

O Capítulo 5 irá apresentar e formalizar os modelos de referência utilizados pelos monitores gerados.

O Capítulo 6 irá apresentar a sintaxe da linguagem bem como sua semântica formal.

No Capítulo 7 serão detalhados os experimentos e, a seguir, o Capítulo 8 concluirá a abordagem expondo os pontos positivos, negativos e o que ainda falta concluir para finalização da tese.

Por fim, no Apêncice A este documento disponibiliza a gramática da TDevC em BNF e, no Apêncice B, este documento disponibiliza as especificações em TDevC utilizadas nos experimentos.

2 FUNDAMENTAÇÃO TEÓRICA

Para que os capítulos que se seguem tenham um melhor entendimento é preciso apresentar alguns conceitos que permeiam a abordagem proposta. O primeiro a ser discutido, na seção seguinte, diz respeito à comunicação entre componentes de uma plataforma e o papel dos HdS, em especial os *device drivers*. Na sequência, na sec:fund:mps, serão abordados temas relacionados a metodologias de projeto de sistemas, tais como projetos baseados em plataformas e em modelos. Por fim, na sec:fund:contract, será apresentado o conceito de projetos baseados em contratos e lógica temporal, que são a base da definição das propriedades suportadas pela abordagem proposta.

2.1 Comunicação entre componentes de plataformas

Elementos de processamento em uma plataforma computacional se comunicam através de suas interfaces. Esta comunicação pode ocorrer de maneira direta, quando as interfaces são conectadas diretamente umas com as outras (ponto-a-ponto), ou, o que é bastante comum, através de um elemento de comunicação compartilhado. Um elemento de comunicação compartilhado, como por exemplo um barramento, representa logicamente os canais de comunicação entre as tarefas sendo executadas nos elementos de computação do sistema conectados a ele.

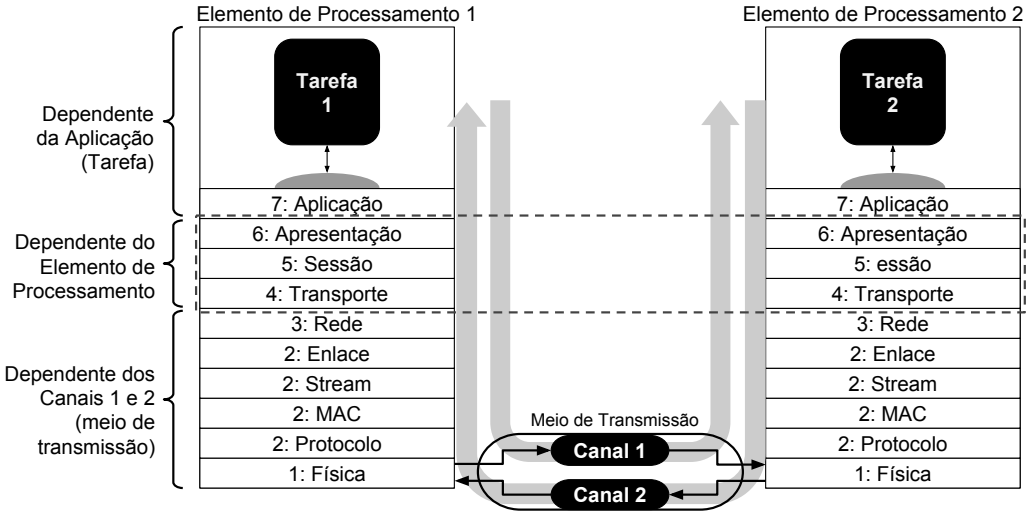


Figura 5 – Escopo da validação do protocolo de comunicação proposto.

Para explicar o que foi dito, a Figura 5 mostra a comunicação entre dois elementos de processamento através de um meio comum de transmissão de dados. Com base em uma adequação proposta por Gajski et al. (2009) da pilha de protocolos do modelo International Organization for Standardization (ISO)/Open Systems Interconnection (OSI) (ZIMMERMANN, 1980) aplicada para a comunicação entre tarefas em sistemas

computacionais, a Figura 5 mostra a ordem das camadas envolvidas e a dependência que elas têm em relação aos elementos do sistema. Como pode ser visto, cada camada na figura tem associado a si um número equivalente à camada no modelo OSI. Por uma questão de detalhamento, Gajski et al. (2009) subdividiu a camada de enlace do modelo OSI em quatro camadas: Enlace, Stream, MAC e Protocolo.

Semelhante às redes de computadores, cada camada presta um serviço para as camadas acima delas e quanto mais abaixo é a camada, maior é a sua dependência com o meio físico de transmissão de informação, ou seja, menor é a sua abstração.

A camada de aplicação é a camada mais próxima da tarefa em execução. Através de funções de comunicação em alto nível de abstração, permitindo inclusive o uso de tipos de dados complexos, esta camada é o início da comunicação entra as tarefas 1 e 2. Esta camada é completamente independente da implementação do outro elemento de processamento, assim como do meio de transmissão. Ela é apenas dependente da tarefa em questão.

Já as camadas de Apresentação, Sessão e Transporte, envolvidas pela linha tracejada na Figura 5, implementam funcionalidades relacionadas à tradução e organização de dados suportados pelo outro elemento de processamento. Estas 3 camadas implementam o protocolo de comunicação de alto nível entre os componentes, o qual é o alvo desta abordagem. Este protocolo é dependente exclusivamente do componente com o qual o elemento em questão está se comunicando, sendo completamente independente do meio de transmissão envolvido.

A camada de Apresentação define a tradução dos dados enviados da camada de aplicação para um formato suportado pelo componente remoto. Esta camada agrupa as informações em grupos de bits e na ordem correta. Esta ordem é justamente a sequência dos dados esperada pelo componente remoto.

Já a camada de Sessão é responsável por unificar os canais de envio e recebimento de dados em um único meio e multiplexar o seu uso para enviar os dados da camada de apresentação, tratar interrupções e solicitar dados ao dispositivo. É através da camada de sessão que há o gerenciamento de vários contextos de comunicação (sessões) entre um mesmo componente remoto e logicamente associados a diferentes canais, mesmo que utilizando um único meio de transmissão.

A camada de Transporte é responsável por dividir os dados enviados e reagrupar os dados recebidos pela/para a camada de Sessão. Esta camada divide os dados em quadros com tamanhos suportados pela arquitetura. Todos os dados enviados desta camada para as camadas de níveis mais baixos são abstraídos pela abordagem.

As camadas abaixo da camada de Transporte englobam o protocolo de comunicação do meio de transmissão (barramentos, Network-on-Chip (NoC), pontes, transdutores, etc.) e não dos dispositivos que agregam as funcionalidades da aplicação à plataforma. Este protocolo atende a todos os dispositivos, sendo o protocolo base para a transmissão de

dados.

Focar nas camadas de Apresentação, Sessão e Transporte possibilita analisar apenas o protocolo de comunicação específico do dispositivo envolvido, sem se preocupar com o protocolo do meio de transmissão sob o qual ele está sendo encapsulado. Este protocolo de alto nível, puramente dependente do componente, é único e é ele que irá ditar como este componente irá se comportar. É neste conjunto de camadas que se encontram os HdS, os quais serão detalhadas na próxima seção.

2.1.1 Software dependente de Hardware (HdS)

Atualmente o projeto de software embarcado tem impacto nos custos de projeto. Até a uma década atrás, apenas 10% dos custos de projetos de sistemas eletrônicos eram decorrentes do software e os outros 90% eram para o desenvolvimento de hardware. Entretanto esse quadro se inverteu (ECKER; MUELLER; DOMER, 2009).

Ecker, Mueller e Domer (2009) mostram que essa mudança no custo do desenvolvimento de hardware e software se deu por dois motivos principais: flexibilidade e configurabilidade. A flexibilidade desejada só pode ser alcançada através do reuso e compartilhamento de recursos de hardware, resultando em uma necessidade de software robusto para controle e uso destes recursos.

O outro aspecto, configurabilidade, diz respeito ao uso de processadores de propósito geral, como por exemplo microprocessadores e Digital Signal Processor (Digital Signal Processor (DSP)). Para redução de custos na manufatura, os processadores são produzidos em grandes quantidades e são aplicados em diversos cenários como telecomunicações, aviação, automobilismo e etc. Logo, é através do software que são feitas as reconfigurações necessárias para a adaptação destes componentes. É importante frisar que as partes de software que são mais críticas em relação à flexibilidade e configurabilidade são as camadas mais ligadas ao hardware, conhecidas como software dependente de hardware (HdS).

Ecker, Mueller e Domer (2009) definem HdS como sendo funções do software embarcado que interagem diretamente com o hardware no qual estas funções estão sendo executadas. Cada HdS é construído especificamente para um determinado recurso de hardware da plataforma, sendo ele completamente desnecessário na ausência deste recurso. Assim hardware e HdS juntos implementam funcionalidades dentro dos sistemas que permitem fazer uso do recurso disponível. Devido a isso, essas funções de software devem ter uma interface simples e clara para as camadas mais superiores.

A Figura 6 mostra uma arquitetura de sistemas onde se destacam claramente os componentes de hardware, as camadas de HdS e o software da aplicação. Na camada acima se encontram as aplicações que são suportadas pelo HdS. No meio se encontram os vários tipos de software dependentes de hardware, servindo de infraestrutura e interface de comunicação entre as aplicações e o hardware.

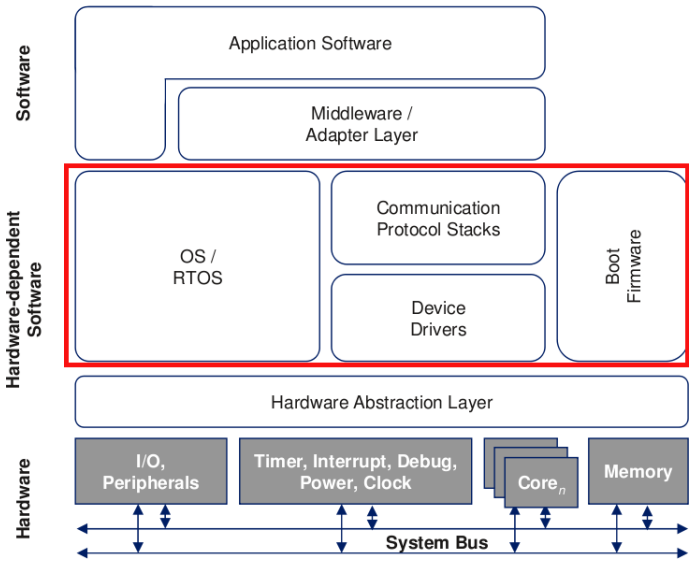


Figura 6 – Contexto geral da arquitetura de sistemas (ECKER; MUELLER; DOMER, 2009)

Ainda de acordo com a Figura 6, os HdS incluem os seguintes tipos de software: Sistema Operacional e Sistemas operacionais de tempo real (Real-Time Operating System (RTOS)), Boot Firmware, Pilhas de protocolos de comunicação, *device drivers* e a Camada de abstração de Hardware (Hardware Abstraction Layer (HAL)). A técnica proposta neste trabalho tem como objetivo suportar o desenvolvimento de *device drivers*. A próxima seção descreverá com mais detalhes esse tipo de HdS.

2.1.1.1 Device Drivers

Device drivers são funções de software responsáveis em prover acesso aos serviços disponíveis pelo hardware. Do ponto de vista das aplicações, os *drivers* podem ser vistos como uma camada de software que abstrai os recursos que o hardware oferece, servindo como uma interface de acesso aos dispositivos e, quando associados a sistemas operacionais, eles podem ser vistos como uma extensão do kernel (SWIFT et al., 2002).

As principais responsabilidades dos *device drivers* incluem suportar a interação com a aplicação e o sistema operacional, a interação com o hardware, realização de alocação de canais de comunicação e gerenciar a utilização concorrente e identificação dos dispositivos. A Figura 7 mostra os diversos cenários em que os *device drivers* podem ser aplicados.

São os *device drivers* que viabilizam a comunicação entre tarefas sendo executadas em diferentes elementos de processamento em uma plataforma computacional, mais especificamente a comunicação entre aplicações sendo executadas em processadores e dispositivos. Dispositivos integrados em uma plataforma são recursos disponíveis para todo o sistema. Entretanto, para utilizá-los é preciso conhecê-los e entender principalmente a forma como eles se comunicam. Dessa maneira *drivers* precisam assumir a tradução e formação de dados, a sincronização entre componentes, bem como a escolha e uso do meio

de transmissão de dados entre os pontos que se comunicam (GAJSKI et al., 2009).

Fazendo uma analogia com a realidade, quando duas pessoas de diferentes idiomas ou com deficiências auditivas e de fala querem se comunicar, quando elas não dominam o idioma ou a linguagem do outro, elas precisam de intérpretes. Estes intérpretes são pessoas especializadas nas linguagens que as duas pessoas dominam. Assim, sua tarefa é escutar o que uma das pessoas quer dizer para o outro, interpretar, traduzir a mensagem e montar a frase que será repassada, verificar se a outra pessoa está prestando atenção e escolher o meio de repassar a mensagem (gestos, fala ou escrita), repassar a mensagem e se certificar de que a outra pessoa entendeu. Numa plataforma computacional, um *device driver* é similar a este intérprete.

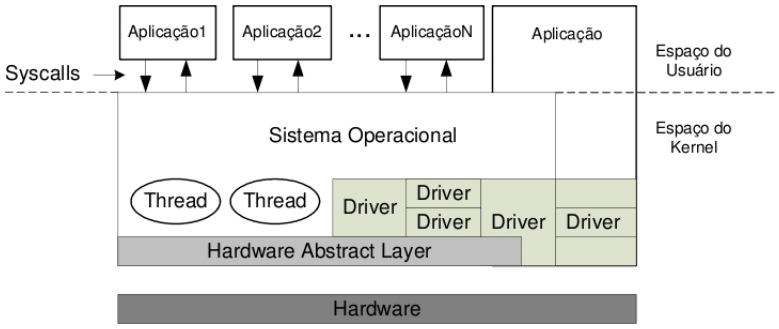


Figura 7 – Cenários em que *device drivers* podem ser aplicados (LISBOA et al., 2009)

Através da Figura 7 é possível visualizar dois cenários principais: *device drivers* utilizados com e sem sistemas operacionais. Sempre que há um sistema operacional o *driver* se encontra em uma parte do *kernel* e, por esse fato, sua execução normalmente é feita em modo *kernel*, apesar de existirem trabalhos, como Ganapathy et al. (2007) por exemplo, que sugerem que partes de *drivers* sejam executas no modo usuário.

Considerados elementos críticos em um sistema eletrônico, os *drivers*, por serem códigos de terceiros e por serem executados pelo kernel do sistema operacional, têm sido uma das maiores causas de falhas de sistemas. Dentre os motivos para este fato estão a complexidade de codificação e verificação, uma vez que seu comportamento é intrinsecamente ligado ao comportamento do hardware (TANENBAUM; HERDER; BOS, 2006; SWIFT et al., 2002).

Assim, faz-se necessária a utilização de técnicas e metodologias que auxiliem os projetistas de *device drivers* a desenvolverem estes HdS de maneira mais rápida e confiável. Algumas metodologias de projeto já vêm sendo largamente utilizadas no desenvolvimento de plataformas computacionais. Porém, como discutido no sec:intro, algumas lacunas ainda precisam ser cobertas, principalmente no desenvolvimento de software embarcado, em especial software dependente de hardware. A seções seguintes irão detalhar sobre estas metodologias.

2.2 Metodologias de Projetos de Sistemas

Há algum anos, empresas de desenvolvimento de sistemas não se davam conta de que metodologias, técnicas e ferramentas de projeto de sistemas eram o meio ideal para reduzir ou solucionar problemas críticos em sistemas computacionais. Porém este quadro mudou (SANGIOVANNI-VINCENTELLI; DAMM; PASSERONE, 2012).

Projetar sistemas computacionais atualmente não é uma tarefa simples. Projetos de sistemas industriais, tais como transporte, saúde e comunicação, estão lidando com dificuldades devido ao aumento significativo de suas complexidades. Aliado a isso, exigências como confiabilidade, diversidade de funcionalidades e *time-to-market* acrescentam mais restrições ao desenvolvimento destes sistemas. A omissão de atenção para qualquer uma dessas exigências pode ser motivo para perda de confiança em um produto ou empresa, além de elevados custos para corrigir tardiamente eventuais erros de projeto (*recalls*, indenizações, etc.).

Desta forma, é necessário o uso de metodologias que suportem o desenvolvimento, verificação e validação dos requisitos funcionais e não funcionais dos sistemas, uma vez que a segurança e confiabilidade deles vem sendo ameaçadas à medida que suas complexidades crescem exponencialmente e requisitos críticos se tornam cada vez mais difíceis de serem observados e avaliados.

Adicionalmente, projetos de sistemas devem levar em consideração não só componentes e sistemas em desenvolvimento localmente, mas sim toda a cadeia de fornecedores da indústria, desde empresas montadoras e de primeiro setor até empresas de fornecimento de peças e subcomponentes. É fundamental que estas empresas interajam de maneira clara e sem equívoco ou más interpretações de requisitos de projetos.

Há algumas décadas, linhas de pesquisa já vêm sendo exploradas para atacar as exigências relacionadas ao desenvolvimento de sistemas face ao crescimento das suas complexidades, apostando no desenvolvimento iterativo e incremental (LARMAN; BASILI, 2003). Algumas dessas linhas de pesquisa bem-sucedidas envolvem técnicas para projetos baseados em camadas, projetos baseados em componentes, projetos baseados em modelos e projetos baseados plataformas, das quais estas duas últimas serão discutidas nas seções que se seguem.

É importante deixar claro que nenhuma dessas metodologias que serão explicadas pode ser considerada melhor que a outra. Elas são comumente aplicadas em conjunto, uma vez que cada uma ataca o problema em um diferente aspecto ou ponto de vista.

2.2.1 Projetos Baseados em Modelos

Modelos são representações em alto nível de componentes de um sistema. Projetos baseados nestas representações suportam as principais metodologias capazes de lidar com a crescente complexidade de sistemas, uma vez que ela suporta a validação prévia de requisitos e a

integração virtual de componentes. Diversas linguagens e ferramentas, como por exemplo, Simulink (BEUCHER, 2006), SystemC (BLACK et al., 2009) e Modelica (TILLER, 2012) suportam a especificação e execução de modelos.

Diferentes modelos, representando diferentes características, são integrados em plataformas virtuais (sec:pbd) de forma a permitir a execução e visualização prévias do sistema. Desta forma é possível analisar atributos como funcionalidades, conectividades, eficiência de algoritmos, comunicação, sincronização, coerência, roteamento e métricas de projeto como desempenho, consumo de energia, restrições temporais, taxa de transferência de dados, etc. (GAJSKI et al., 2009).

O uso de modelos proporcionou o desenvolvimento de projetos de maneira incremental, onde cada modelo é especificado em um alto nível de abstração (modelos simples) e estes são refinados para modelos mais detalhados em níveis mais baixos de abstração com a adição de detalhes mais próximos do componente real. Dessa forma, cada modelo em um certo nível de abstração serve como uma especificação do modelo que será refinado no nível de abstração abaixo (GAJSKI et al., 2009). Este tipo de metodologia é conhecido como *Specify-Explore-Refine* (Specify-Explore-Refine (SER)) (GAJSKI et al., 1994; JANKA, 2002).

Em meio a diversas discussões sobre qual a quantidade ideal de níveis de abstração modelados para se especificar um sistema, (GAJSKI et al., 2009) propõe que a modelagem dos diversos níveis de abstração seja baseada nos três tipos de projetistas envolvidos no desenvolvimento do sistema: (i) **Projetista de aplicação**, o qual tem conhecimento da aplicação, sua estrutura e algoritmos, mas apenas uma noção simples do sistema como um todo. (ii) **Projetista de Sistema**, o qual tem um bom conhecimento em arquitetura e organização de sistemas, em arquitetura e operação de processadores, barramentos, etc., mas tem muito pouco conhecimento da aplicação e de técnicas de implementação de componentes. Por fim, (iii) **projetista de implementação**, o qual é especialista em componentes específicos e em técnicas, ferramentas e metodologias de implementação destes componentes.

A partir desses perfis de projetistas foram definidos pelo menos três níveis de abstração básicos. São eles:

- **Modelo de especificação**: são modelos simplificados onde basicamente são implementados algoritmos representando tarefas e comunicação entre elas, independentemente de qualquer plataforma alvo. É usado por projetistas de aplicação para desenvolver, validar e ajustar os algoritmos (tarefas), usados na realização da aplicação, e seus canais de comunicação (GAJSKI et al., 2009).

- **Modelo em nível de transação (Transaction-level Modeling (TLM))**: são modelos com uma representação simplificada da plataforma, mas com a alocação das tarefas nos elementos de processamento. Mesmo sendo algo mais próximo de uma plataforma computacional, ainda não se tem nenhuma informação concreta sobre

tempo e sinais digitais. A troca de informação entre componentes ocorre através transações de alto nível por meio de chamadas de funções. Com este modelo já é possível estimar métricas como desempenho, processamento, taxa de comunicação, consumo de energia e confiabilidade dos componentes. Como é usado por projetistas de sistemas, através deste modelo o projetista pode explorar diversas formas de implementação tanto de hardware (meios de comunicação de componentes, processadores, etc.) como de software (sistemas operacionais, *firmwares*, *device driver*, etc.) (GAJSKI et al., 2009).

- **Modelo com precisão de ciclo (Cycle-Accurate Model (CAM))**: são modelos mais próximos da plataforma real e que agregam informações mais concretas dos componentes como temporização e sinais. Este modelo é usado tanto por projetistas de implementação de hardware (na especificação de processadores de aplicação específica (Application-Specific Instruction Set Processor (ASIP)), controladores de dispositivos como memória e interrupção, barramentos, pontes, etc.) como por desenvolvedores de software (na definição de pontos de sincronização, definição de escalonamento, definição de largura de banda, definição de interrupções. etc.) (GAJSKI et al., 2009).

Estes modelos são usados nas diferentes fases da implementação de sistema. Eles são modelos complexos que constituem plataformas virtuais executáveis, como podem ser vistos na Figura 8.

Um projeto de um sistema começa com a definição dos requisitos da aplicação que este vai atender. Estes requisitos definem as regras da aplicação e definem o que deve ser feito e de que maneira. A partir destes requisitos um modelo de computação (Model of Computation (MoC)) é especificado. MoC é uma maneira generalizada de descrever comportamentos abstratos e conceituais de sistemas. É uma maneira simplificada de modelar o comportamento (requisitos e restrições) da computação e comunicação das tarefas (GAJSKI et al., 2009).

A partir de MoC, programadores implementam modelos de especificação, os quais são materializações dos modelos de computação. A partir destes modelos já é possível validar a funcionalidade da aplicação e ter uma ideia inicial da plataforma computacional.

Sendo viável, a partir da análise de modelos de especificação define-se uma arquitetura de uma plataforma base. É neste ponto que o conceito de PbD entra em ação. Neste momento, componentes em alto nível de abstração compõem uma plataforma virtual extremamente simplificada e sem modelagem de tempo. O projetista analisa quais componentes melhor se encaixarão na plataforma para atingir os requisitos da aplicação. Para isso uma versão simplificada do software já pode ser desenvolvida e executada em modelos simples de processadores conhecidos como simuladores de conjunto de instruções (Instruction Set Simulator (ISS)), os quais se comunicam com os modelos de controladores de dispositivos

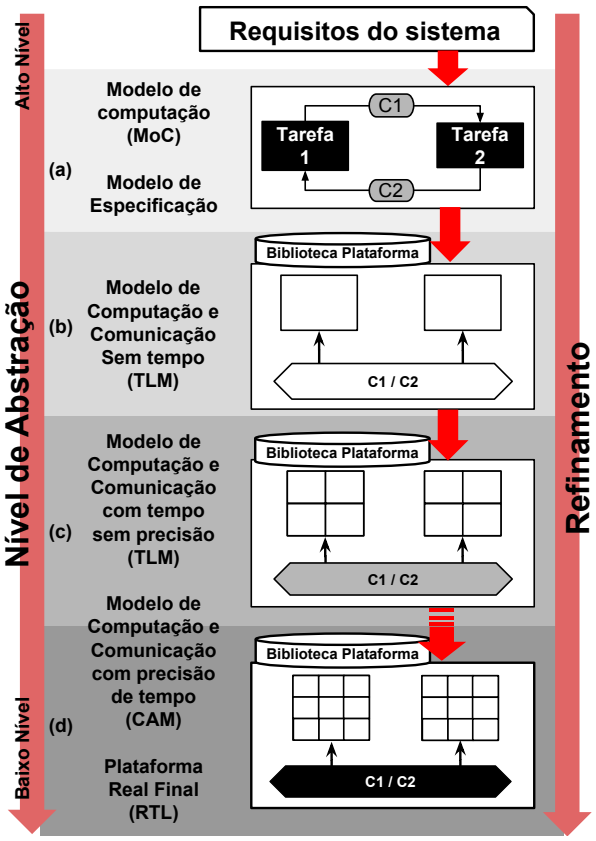


Figura 8 – Processo incremental de projetos de sistemas

através de transações em alto nível baseadas em modelos de componentes em nível de transação (TLM) (GHENASSIA et al., 2005; CAI; GAJSKI, 2003).

Simulando e analisando a plataforma base, uma série de decisões devem ser tomadas, com por exemplo, escolhas de componentes e elementos de comunicação, ajustes de modelos, etc. Numa plataforma inicial, basicamente valida-se a funcionalidade dos componentes.

A medida que se refina a plataforma, modelos com informação de temporização são associados aos componentes e à comunicação, porém sem precisão de ciclo (*clock*). Seguindo os refinamentos e incorporando mais detalhes aos componentes, adiciona-se uma referência de tempo e consequentemente associa-se tempo com precisão de cíclo de relógio através de modelos de componentes e barramentos com precisão de ciclo (CAM / Bus Cycle-Accurate Model (BCAM)).

Por fim, com todos os modelos definidos e o software basicamente definido e implementado, gera-se o modelo em nível de transferência de registrador (Register Transfer Level (RTL)) da plataforma, o qual pode ser sintetizado em *netlists* para FPGA ou para posterior síntese de *layout* e prototipação como Application Specific Integrated Circuit (ASIC) em silício.

A seção seguinte irá detalhar a técnica de projetos baseados em plataformas, uma metodologia que se baseia no uso dos modelos de plataformas predefinidas para desenvolver sistemas computacionais suportando melhor decisão de uso de recursos e reduzindo o

tempo de projeto.

2.2.2 Projetos Baseados em Plataforma

O principal objetivo de projetos de sistemas é encontrar o equilíbrio entre tempo e custo de projeto com base em restrições de desempenho e funcionalidades (SANGIOVANNI-VINCENTELLI; MARTIN, 2001). A pressão do mercado exigindo tempos de projetos cada vez mais curtos, a complexidade dos dispositivos dificultando a validação de propriedades críticas, juntamente com o elevado custo de manufatura de componentes de hardware, exigem estilos e metodologias de projetos cada vez mais organizados e disciplinados (SANGIOVANNI-VINCENTELLI, 2002).

Com aumento nos custos de manufatura de componentes de hardware, o desenvolvimento de circuitos integrados sentiu a necessidade de migrar de projetos full-custom para projetos onde, a partir de elementos pré construídos, fosse possível realizar uma montagem rápida de SoC. Com esse tipo de abordagem, surge o foco em componentes de propósito geral ou reconfiguráveis. Essa necessidade de se ter projetos de sistemas embarcados mais flexíveis levou a indústria de eletrônicos na direção de projetos baseados em soluções programáveis e reorganizáveis, impulsionando o foco para metodologias, infraestruturas e ferramentas que seguem esta linha.

Projetos baseados em plataformas têm como base uma metodologia que ameniza os problemas relacionados ao custo de manufatura e à produtividade de projetos, fundamentando-se na flexibilidade e reuso de componentes. Como o próprio nome diz, esta técnica se baseia no conceito de plataforma como pontos de articulação para representar as camadas de abstração entre as fases no fluxo do projeto.

Conceitualmente, uma plataforma pode variar de acordo com o domínio da aplicação. Entretanto, para PbD uma plataforma nada mais é do que uma biblioteca de componentes que podem ser usados para gerar uma solução em cada nível de abstração.

Os princípios desta metodologia são:

- Utilização de modelos alto nível de abstração para esconder detalhes desnecessários de implementação, como mostrado na seção anterior.
- Utilização de uma biblioteca de componentes previamente implementados com o objetivo de restringir o espaço de exploração.
- Alcançar uma implementação, a partir de uma especificação inicial, seguindo de uma sequência de refinamentos, utilizando modelos mais detalhados de uma plataforma em cada passo.

Como mostra a Figura 9, os componentes da plataforma possuem parâmetros de configuração que garantem a flexibilidade do método, possibilitando seu refinamento para encontrar uma solução adequada. A seleção de um conjunto de componentes com

a definição dos parâmetros de configuração resulta em uma solução que é chamada de instância de plataforma.

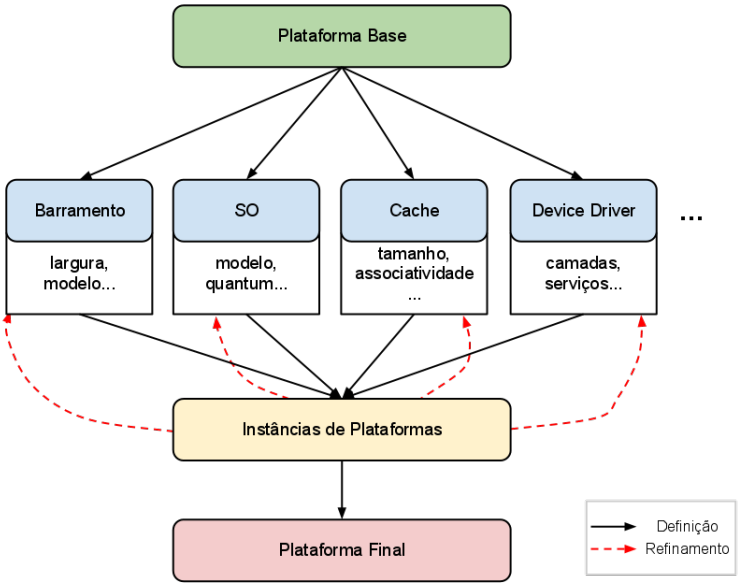


Figura 9 – Dinâmica de projetos baseados em plataformas

Muitos estudos estão sendo dedicados ao desenvolvimento de técnicas e ferramentas que auxiliam no mapeamento de componentes de plataformas virtuais para componentes reais. Como citado no sec:intro, trabalhos relacionados à geração de *device drivers* focam nos desafios desta metodologia e tentam desenvolver técnicas de síntese de HdS com o intuito minimizar o esforço no mapeamento de *drivers* em arquiteturas alvo.

Mesmo utilizando metodologias robustas e que permitem uma integração de componentes e seus modelos de maneira bem simplificada, algumas preocupações continuam existindo do ponto de vista de integração de componentes (acoplamento "horizontal"), de fidelidade de componentes em relação às suas especificações implementadas a partir de modelos em altos níveis de abstração e interfaceamento de diferentes camadas de sistemas (acoplamento "vertical").

Para se ter métricas obtidas de maneira confiável em uma plataforma virtual é necessário que todos os seus componentes/modelos respeitem todos os requisitos exigidos tanto pela especificação da aplicação quanto pela especificação do componente em si. Assim, a seção a seguir irá apresentar uma metodologia de projetos com base em asserções na forma de contratos que definem garantias e suposições de comportamentos das implementações de componentes e do meio no qual estes estão inseridos.

2.3 Projeto Baseado em Contratos

Esta metodologia de projetos tem como fundamento a aplicação de contratos entre componentes para garantir que cada um esteja cumprindo com a suas obrigações. A

definição de contrato no dicionário Aurélio (FERREIRA, 1986) é bem clara e se encaixa perfeitamente para o que está se propondo:

> *Contrato: "Acordo ou convenção para a execução de algo sob determinadas condições."*

Exatamente o que diz essa definição, a aplicação de contratos entre componentes define um acordo nas garantias de sua implementação com base em supostas condições providas pelo ambiente no qual este componente está inserido.

Estes contratos especificam **garantias** de uso das interfaces de componentes com base em **suposições** de uso dessas mesmas interfaces por parte do ambiente. Assim, contratos são definidos através de tuplas, onde um contrato C qualquer é definido pela equação $C = (A, G)$, onde A representa as condições do contrato (suposições) e G representa as obrigações ou acordos (garantias) do contrato.

As suposições e garantias são asserções que representam sequências de acessos (*traces*) às interfaces dos componentes. Assim, quando se tem um contrato associado à especificação de um componente f qualquer, se diz que uma implementação M_f deste componente **satisfaz** um contrato C_f (escreve-se $M_f \models C_f$), quando $M_f \cap A_f \subseteq G_f$ (D'ANGELO, 2014; SANGIOVANNI-VINCENTELLI; DAMM; PASSERONE, 2012). Esta expressão significa que a satisfação de C_f por M_f implica que os *traces* gerados por M_f, quando respeitados pelo ambiente através de suas suposições A_f, formam ou são os *traces* considerados corretos por suas garantias G_f. Em outras palavras, quando há respeito de A_f pelo ambiente, então a implementação M_f respeita as garantias descritas em G_f,

Porém, existem situações em que A_f pode não ser respeitado. Isto não implicaria dizer que a implementação não satisfaz o contrato C_f, porém implica dizer que não há **compatibilidade** entre a implementação M_f e o ambiente. Assim, pode-se dizer que, quando há **compatibilidade** de um componente com seu ambiente, este deve respeitar suas garantias para **satisfazer** o contrato.

Como dito no final da seção anterior, contratos podem ser aplicados em acoplamentos "horizontais"e "verticais". Os acoplamentos "horizontais" compõem as integrações de modelos ou componentes. Assim, com base em suas interfaces de comunicação, os contratos "horizontais" especificam *traces* dessa comunicação que definem a compatibilidade dos componentes e, quando ela existe, a satisfazibilidade do componente integrado deve ser garantida pela sua própria implementação (BENVENISTE et al., 2011).

Já os contratos "verticais" definem suposições e garantias em subcomponentes especificados e modelados a partir do refinamento de sistemas ou componentes em camadas em um nível de abstração maior. Assim, estes contratos garantem que o subcomponente que agora faz parte de um componente maior é compatível neste ambiente e que, com a presença do mesmo, o componente o qual foi modularizado continua respeitando as garantias do contrato (BENVENISTE et al., 2011).

A abordagem proposta nesta tese está baseada nestes dois tipos de contratos no acoplamento entre *device drivers* e seus dispositivos. O foco desta proposta neste acoplamento é em relação ao protocolo de comunicação de alto nível entre componentes em uma plataforma, encapsulando e tornando completamente transparente para o projetista de sistema o protocolo do meio de transmissão dos dados.

Contratos "horizontais" irão garantir que as propriedades do protocolo estão sendo respeitadas em uma certa camada de abstração. Em contrapartida, a composição de contratos "horizontais", adicionados aos refinamentos, formam os contratos "verticais", garantindo que, mesmo refinadas, as propriedades mais abrangentes continuam sendo respeitadas.

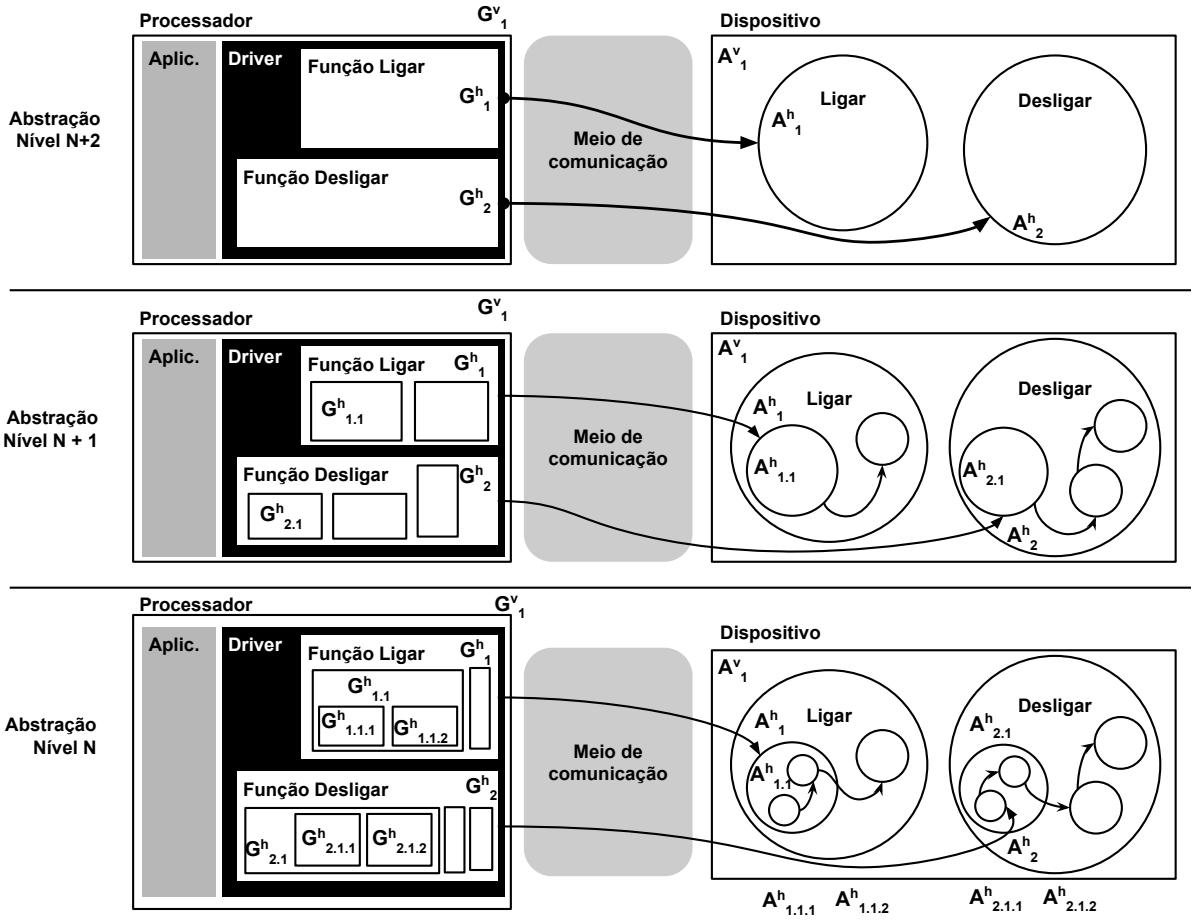


Figura 10 – Exemplo mostrando a aplicação da metodologia baseada em contratos.

A Figura 10 mostra um exemplo prático da aplicação de contratos no desenvolvimento de *device drivers*. Como pode ser visto na Figura 10, de maneira horizontal há um processador e um dispositivo se comunicando via um meio de comunicação qualquer. De maneira vertical, é possível perceber que há 3 níveis de abstração de modelos, tanto do *device driver* quanto do dispositivo. São os níveis N+2, N+1 e N, onde o nível N+2 é o nível

de maior abstração e o N o nível de menor abstração e o nível final do desenvolvimento do sistema.

Inicialmente, um contrato foi especificado com o intuito de definir regras que respeitam os requisitos de fabricação e uso do dispositivo. Pode-se chamar esse contrato de $C_1^v = (A_1^v, G_1^v)$, onde A_1^v representa as suposições do dispositivo e G_1^v as garantias de que o *device driver* deve respeitar para fazer uso correto do dispositivo, se este último respeitar as suposições A_1^v. Suponha que a suposição A_1^v seja, por exemplo, de que "o dispositivo, após ser inicializado, estará pronto para ser operado após 20 ms". Assim, a garantia G_1^v será a de que "o *device driver* nunca irá requisitar alguma informação ou nunca irá enviar comandos ao dispositivo antes de 20 ms após sua inicialização". Esta contrato deve ser respeitado durante todas as fazer do desenvolvimento do dispositivo, independente do nível de abstração. Pode-se dizer que este é um contrato vertical.

Como também pode ser visto na Figura 10, outros contratos foram definidos. Pode-se chamar esses contratos de $C_1^h = (A_1^h, G_1^h)$ e $C_2^h = (A_2^h, G_2^h)$, onde, respectivamente, definem regras para uso da interface do dispositivo. Em outras palavras, o contrato C_1^h define que a implementação do *device driver* deverá respeitar as garantias G_1^h para conseguir "ligar"o dispositivo e deverá respeitar as garantias G_2^h, para "desligar"o dispositivos, supondo que o dispositivo irá funcionar de acordo com as suposições G_1^h e G_2^h, respectivamente.

É possível perceber ainda na Figura 10 que, à medida que o nível de abstração diminui, outras suposições e garantias fazem parte da composição dos contratos C_1^h e C_2^h. Isso acontece porque o processo de desenvolvimento baseado em plataforma permite o refinamento e o incremento dos modelos. Assim, quando detalhes são adicionados, a complexidade das funcionalidades aumenta, mais suposições de comportamentos de sub-módulos do dispositivo precisam ser consideradas e mais garantias para o correto uso destes sub-módulos podem ser necessárias.

Um detalhe que é interessante ser destacado é o fato de que as suposições e garantias A_1^h, G_1^h, A_2^h e G_2^h acabam compondo também contratos verticais, uma vez que estas suposições são validas durante todo o processo de desenvolvimento, bem como as garantias de uso. Este ponto reforça uma questão importante lavada em consideração nesta técnica proposta: A checagem de contradições entre as garantias que compõe contratos. Par que garantias especificadas durante o refinamento/incremento de funcionalidades ao logo do desenvolvimento não prejudiquem garantias verticais, este trabalho desenvolveu uma checagem de contradições entre as garantias, verificando se as garantias especificadas em sub-módulos invalidam as garantias mais antigas (especificadas nos modelos nos níveis mais altos de abstração).

2.3.1 Propriedades em Lógica Temporal Linear

As propriedades que representam contratos, discutidas anteriormente, precisam ser especificadas de uma maneira que seja possível expressar comportamentos desejados com

base nos elementos e nas variáveis que fazer parte da implementação do sistema na qual este contrato é atuante. A lógica de primeira ordem é uma ferramenta fundamental para construção dessas propriedades, entretanto, ela indica a veracidade de uma expressão de maneira momentânea, não importando o que aconteceu antes ou o que acontecerá no futuro.

Em situações onde se deseja ter um contexto temporal (stateful) do comportamento de um programa ou sistema, se faz necessária a aplicação da lógica, porém com o conhecimento da sua veracidade em momentos anteriores (KRöGER, 1987; ØHRSTRØM; HASLE, 1995).

Para atender essa demanda, a lógica modal, especificamente a lógica temporal, é utilizada para descrever sequências de estados em sistemas reativos (LETTNIN, 2009). Em outras palavras, através da lógica temporal é possível descrever premissas cuja veracidade é sensível ao longo do tempo, com base em eventos temporais, ou seja, eventos que ocorrem ao longo do tempo, porém sem necessária explicitar o valor temporal (Ex.: segundos, pingar de uma torneira, acionamento de um botão em um elevador, etc.).

A lógica temporal pode ser ramificável ou linear. Lógica ramificável, entretanto, pode dissertar sua veracidade com base em múltiplas linhas temporais. Como exemplo para lógica ramificável pode-se definir uma expressão que "existe a possibilidade que chova para sempre. Pode-se dizer também, por exemplo, que "existe a possibilidade que eventualmente não chova mais.". Como não há um certeza, essas duas premissas podem ser verdadeiras em certos momentos.

A lógica temporal linear está restrita a dissertar a veracidade em uma única linha no tempo, não levando em conta valores probabilísticos. Como exemplo para lógica linear pode-se definir uma expressão que "sempre chove às 12:00 hrs". Para a primeira premissa, "chover às 12:00 hrs" deve acontecer sempre para que ela seja verdadeira. Caso um único dia não "chova às 12:00 hrs", a premissa passa a ser falsa.

Essas premissas podem ser representadas a partir de fórmulas Linear Temporal Logic (LTL), onde estas representam propriedades temporais ao longo de tempo de maneira infinita, ou seja, não possuem um momento final. Tais propriedades são descritas com a ajuda de operadores que definem proposições lógicas com referência a algum ponto no tempo (KRöGER, 1987). Assumindo-se que a e b são premissas e i é uma referência ao evento corrente, os principais operadores são mostrados a seguir:

- **X(m) a**: operador de próxima ocorrência. Este operador indica que a deve ocorrer no evento i+m.
- **G(m,n) a**: operador global. Este operador indica que a deve ocorrer sempre no período $m \leq i \leq n$.
- **F(m,n) a**: operador de eventualidade. Este operador indica que a deve ocorrer em algum momento no período correspondente a $m \leq i \leq n$.

- **a A b**: operador next. Este operador indica que a deve ocorrer após a ocorrência b.
- **a U b**: operador until. Este operador indica que a deve sempre ocorrer até que b ocorra.

Tendo como referência o exemplo "sempre chove às 12:00 hrs" e associando "chove" à letra a e 12:00 hrs à letra b, podemos escrevê-lo da seguinte maneira: $G(a \ \& \ b)$.

As expressões LTL podem ser representadas através de autômatos. Para se ter uma aplicação computacional, esta tese a representa através de Autômatos Büchi (GASTIN; ODDOUX, 2001). Autômatos Büchi, de maneira sucinta, são máquinas de estados que, para ter a linguagem aceita, bastam que algum estado final tenha sido alcançado pelo menos uma vez.

Este tipo de representação funciona bem para a validação de propriedades de contratos, uma vez que para se identificar uma situação na qual uma propriedade não é válida, basta o Autômatos Büchi (a propriedade) aceite o trace da violação da propriedade em algum momento da execução.

2.4 Resumo

Este capítulo fez um levantamento da principal fundamentação teórica na qual esta abordagem se baseia. Inicialmente foi feita uma apresentação sobre software dependente de hardware, especialmente os *device drivers*. Em seguida foi explicado sobre metodologias de projetos de sistemas, onde consequentemente também entra o desenvolvimento de HdS. Nessa seção foram apresentadas as principais metodologias utilizadas pela indústria. Foi dado destaque também à metodologia de projetos baseados em contratos e, por fim, foi apresentado o formalismo da lógica temporal através das fórmulas LTL, as quais são bastante utilizadas nas definições de propriedades envolvidas nos contratos.

3 TRABALHOS RELACIONADOS

Como mencionado nos capítulos anteriores, a comunicação entre processadores e periféricos em uma plataforma é algo extremamente crítico. É fundamental garantir que a troca de informações entre estes componentes em uma plataforma esteja sendo feita de maneira correta.

Diferentes maneiras de olhar para o problema resultam em diferentes métodos e técnicas com o objetivo de identificar falhas ou lidar com sua presença, e melhorar assim a dependabilidade de sistemas embarcados (AVIZIENIS et al., 2004).

Tais métodos ou técnicas propõem impedir ou reduzir a presença, surgimento ou a propagação de falhas e seus efeitos, afim de garantir a correta construção e execução de sistemas. Inicialmente tenta-se evitar desenvolver ou usar componentes com erros, entretanto, para projetos complexos e de grande porte, as metodologias atuais de desenvolvimento e verificação não conseguem garantir 100% de sua corretude.

Um dos grandes desafios no desenvolvimento e verificação de sistemas é detectar e identificar as falhas. A verificação formal realizada através de *model checkers* com base em modelos é bastante poderosa, entretanto, para sistemas grandes e complexos, são quase impraticáveis devido ao tempo e recursos consumidos (BAIER; KATOEN, 2008; LETTNIN et al., 2009). Além disso, como mesmo citou Baier e Katoen (2008): "*Any verification using model-based techniques is only as good as the model of the system.*". Em suma, *model checkers* dependem de um modelo bem feito do sistema, mas a sua implementação real, por diversos motivos, pode divergir deste modelo.

Em contrapartida, técnicas de verificação semiformal através do uso de formalismos aplicados em conjunto com simulação vêm se mostrando bastante eficientes. Metodologias com base em verificação ou validação em tempo de execução aproveitam o fluxo da execução da plataforma para "observar" certas propriedades formalmente especificadas (LETTNIN et al., 2009; BEHREND et al., 2011; BALASUBRAMANIAN; LOWRY; CORINA, 2011; REINBACHER et al., 2012; PIKE; NILLER; WEGMANN, 2012; ZHI-BING et al., 2012; BEHREND et al., 2014; VILLARRAGA et al., 2014).

Entretanto, quando estes sistemas são desenvolvidos por equipes diferentes, verificar ou validar seus componentes isoladamente nem sempre é o principal problema. Integrar componentes desenvolvidos por diferentes empresas passa a ser o grande desafio, uma vez que eles são implementados com base em especificações informais, dando margem a diferentes interpretações. A formalização coerente de requisitos de sistemas e descrições de componentes é um dos atuais desafios enfrentados por pesquisas em desenvolvimento de sistemas embarcados modernos (SANGIOVANNI-VINCENTELLI; DAMM; PASSERONE, 2012).

Diversas metodologias com a de projetos baseados em plataformas (PbD), projetos baseados em modelos (MbD) e projetos baseados em componentes (CbD) dão suporte à im-

plementação de plataformas complexas de maneira incremental, apostando na simplificação e partição de seus componentes. Porém, tais metodologias ainda não são totalmente adequadas para garantir que componentes estejam funcionando conforme a especificação, seja no caso de interfaceamento, funcionalidade, desempenho ou segurança. Desenvolvimento de projetos com base em contratos é atualmente bem aplicado em sistemas embarcados para tentar suprir problemas relacionados à integração e às obrigações de componentes em uma plataforma computacional (SANGIOVANNI-VINCENTELLI; DAMM; PASSERONE, 2012; DAMM et al., 2011; FERRANTE et al., 2014a).

No escopo de *device drivers*, sua integração no ambiente vai além de interfaceamento com sistema o operacional e com a camada de abstração de hardware (HAL). *Device drivers* precisam respeitar protocolos implementados por seus respectivos dispositivos. Estes protocolos de comunicação nem sempre são tão claros e diretos, e normalmente são uma porta de entrada para quebra de segurança, tanto física quanto cibernética. Logo o projeto de *device drivers* também deve levar em conta a corretude de suas interfaces e de seu protocolo.

As técnicas de desenvolvimento, validação e verificação mencionadas anteriormente também podem ser aplicadas para o desenvolvimento de *device drivers*, porém, ou se preocupam com a pilha de software como um todo ou se preocupam com o interfaceamento físico de componentes, o que torna difícil a validação e verificação de protocolos de comunicação de alto nível entre *drivers* e periféricos.

Para tentar suprir estas necessidades no desenvolvimento de *drivers*, inúmeras abordagens apostam na geração automática de HdS a partir de modelos simplificados em alto-nível de abstração baseados em suas especificações informais ou requisitos. A técnica conhecida como corretude por construção (*Correctness-by-Construction*) é utilizada para diminuir a inserção de erros e consequentemente a ocorrência de falhas em sistemas computacionais através da redução de codificação manual do software. Várias técnicas propõem este tipo de abordagem (KATAYAMA; SAISHO; FUKUDA, 2000; ZHANG; ZHU; CHEN, 2003; CONWAY; EDWARDS, 2004; BOMBIERI et al., 2009; LISBOA et al., 2009; RYZHYK et al., 2009; ACQUAVIVA et al., 2013).

Uma grande desvantagem deste tipo de técnica quando aplicada a *drivers* está no fato de que eles não são gerados completamente, uma vez que estas componentes são interfaceados com funções de software de alto nível de diversos tipos de sistemas operacionais e ao mesmo tempo são interfaceados com componentes de hardware em um baixo nível de abstração. Isso cria uma dependência muito grande entre estes pontos de interfaceamento, o que torna a codificação do *device driver* altamente específica. Logo, mesmo gerando alguma parte deste componentes de software, a necessidade de codificar ou modificar alguma outra parte do software gerado vai de encontro ao objetivo da técnica e pode ser uma fonte de inserção erros.

Apesar de existirem diversas técnicas para desenvolver, validar e verificar se um

sistema apresenta algum comportamento indesejado, é impossível, devido ao tamanho do espaço de projeto, verificá-lo 100%. Portanto, alguns erros continuam no sistema, mesmo durante a sua fase de operação. Em alguns casos, através de atualizações corretivas, manutenções ou *recalls*, é possível eliminar determinados erros mesmo quando o sistema já está operando, porém, além de extremamente custoso, a necessidade de eliminar erros nesta fase pode diminuir a confiabilidade e o respaldo tanto do sistema quanto do seu fabricante, principalmente quando há uma falha aparente, como visto no sec:intro.

Logo é fundamental que sistemas lidem com falhas em sua execução, ou seja, que eles sejam resilientes. Algumas técnicas, como as propostas por (SWIFT et al., 2002; GANAPATHY et al., 2007; HERDER et al., 2007; HOOLE; TRAORE, 2008; KADAV; RENZELMANN; SWIFT, 2009; WEGGERLE et al., 2011), focam na tolerância a falhas de *device drivers*, isolando-as e tomando controle do sistema, levando-o para um estado seguro. Apesar de ser eficaz, este tipo de técnica comumente gera uma sobrecarga (*overhead*) considerável na execução do sistema quando implementado puramente em software. Isso geralmente acontece porque, para detecção de falhas, todos os pontos de entrada do *device driver* devem ser avaliados e, para o isolamento da falha, partes do *driver* devem ser executadas em modo usuário (*user-mode*). Assim, a análise das mensagens trocadas com os *device drivers* juntamente com as trocas constantes de modo de execução, executando simultaneamente com todo o sistema, geram uma sobrecarga considerável.

Para evitar este tipo de problema, sistemas julgadores (AVIZIENIS et al., 2004) podem trabalhar em conjunto com os sistemas principais, detectando as falhas, tomando ações corretivas necessárias (técnica conhecida como *steering* (REINBACHER et al., 2012; EN, 2003)) ou simplesmente informando a um sistema remoto ou a um responsável sobre a ocorrência do evento.

Abordagens como as de Ferreira (2016), Ferreira e Vargas (2015) propõe uma técnica de detecção de ataque de *Buffer Overflow*. Apesar desta abordagem não se tratar explicitamente de uma técnica de validação de propriedades em sistemas, ela propõe um monitoramento do *pipeline* de um processador com o intuito de identificar, de maneira não intrusiva, quando há um ataque do tipo *Buffer Overflow*. Isso permite que ela seja vista e comparada a técnicas que validaem propriedades, uma vez que o ataque de *Buffer Overflow* é definido por uma série de comportamentos (semelhante a propriedades temporais) que, quando identificados, permitem que a abordagem inicie uma reação para recuperar o sistema levando-o ao último estado válido.

Esta abordagem serve como uma referência para a aplicação de um *watchdog* implementado como módulos de hardware para monitorar situações críticas em tempo de execução de dispositivos já na fase de operação.

Outra abordagem que segue esta mesma linha é proposta por Silva, Bolzani e Vargas (2011). Silva propõe uma técnica para monitoramento baseado em hardware de tarefas em sistemas de tempo real. Este monitor identifica o não cumprimento do deadline de tarefas

em execução.

A seguir, este documento irá apresentar com mais detalhes algumas abordagens que dão suporte a técnicas e metodologias atuais de desenvolvimento e validação de componentes de um sistema, dentre eles *device drivers*. No final deste capítulo há uma análise comparativa entre estes trabalhos relacionados apresentados. Estas abordagens foram classificadas em abordagens baseadas em contatos técnicas, abordagens baseadas em técnicas de validação em tempo de execução e, dentro desta última classificação, técnicas com base em validação de software embarcado e HdS.

3.1 Técnicas Baseadas em Contratos

A seguir serão apresentadas duas técnicas fundamentadas na metodologia de projetos baseados em contratos, onde são definidos acordos ou regras que vinculam implementação de componentes a seus requisitos.

3.1.1 Nuzzo 2013

Nuzzo, em Nuzzo et al. (2013), propõe uma metodologia de projetos baseados em plataforma (PbD) que permite lidar com a complexidade e heterogeneidade de CPS através de contratos de suposições e garantias para formalizar o processo de desenvolvimento hierárquico e composicional de protocolos de controle, sugerindo ainda uma metodologia de síntese de possíveis candidatos a componentes em baixo nível de abstração.

As principais contribuições desta abordagem são apresentar uma nova metodologia de uso de contratos na integração de *frameworks* heterogêneos de análise e modelagem de componentes para síntese, otimização e verificação de sistemas de controle de CPS, em particular sistemas de controle reativos, e demostrar a sua viabilidade através de um caso de uso industrial real, aplicando a metodologia em um projeto de controle supervisor de um sistema eletrônico de energia de aeronaves.

Dada uma infraestrutura da arquitetura, o objetivo desta abordagem é encontrar um controlador que satisfaça um conjunto de requisitos do sistema como um todo ou indicar que não há um controlador viável com esse conjunto de requisitos.

Nuzzo define os requisitos *top-level* do sistema como um contrato $C_S = (A_S, G_S)$, baseando-se na mesma teoria descrita no sec:fund, sec:fund:contract, onde A_S representa o conjunto de comportamentos aceitáveis que o ambiente (infraestrutura física) pode realizar (suposições) e G_S o conjunto de comportamentos que definem os requisitos do sistema (garantias).

Como pode ser visto na Figura 11, tanto os requisitos *top-level* quanto os comportamentos do ambiente podem ser descritos em duas linguagens formais: tanto em LTL Kröger (1987) quanto em Signal Temporal Logic (STL) Maler e Nickovic (2004).

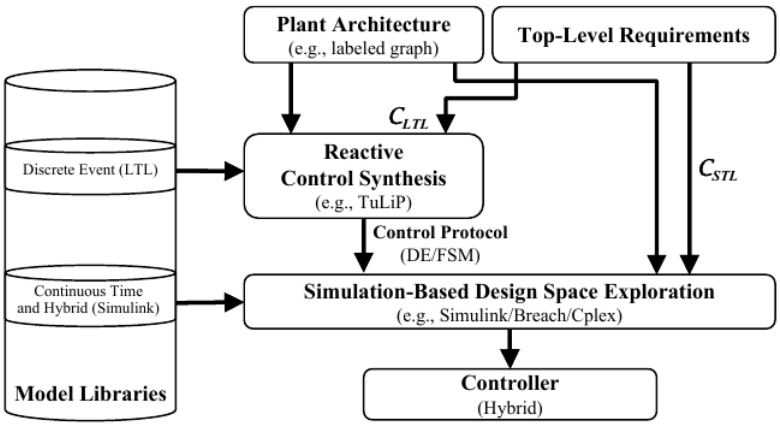


Figura 11 – Fluxo da abordagem proposta por Nuzzo. Fonte: (NUZZO et al., 2013)

A linguagem LTL permite especificar comportamentos temporais caracterizados por lógica Booleana, sinais de tempo discretos e sequência de eventos. Já a STL é utilizada para lidar com modelos dinâmicos híbridos, relacionando modelos discretos com modelos contínuos da parte física da especificação. Os contratos C_{LTL} descrevem requisitos e comportamentos discretos do sistema e do ambiente, e são descritos em LTL. Já os C_{STL} representam os requisitos e comportamentos contínuos ou híbridos, e são descritos em STL.

É possível ainda ver na mesma figura que a metodologia propõe bibliotecas de modelos em 2 níveis de abstração: a camada de eventos discretos (*discrete-event*, DE) e a híbrida (*hybrid*). Os componentes da camada DE são descritos também em LTL e estão em um nível de abstração acima da camada híbrida. É possível ver na Figura 11 que a descrição dos componentes discretos, combinados com os requisitos do sistema e comportamentos do ambiente são utilizados para a síntese dos modelos de controle reativos.

Um componente discreto da camada DE pode ser descrito como um conjunto de variáveis Booleanas e seu comportamento representado através de um *trace* σ, como uma sequência de estados no formato $\sigma = s_1 s_2 s_3 ...$ onde cada estado s corresponde à veracidade das variáveis em um momento do *trace*.

A síntese do controlador reativo é puramente baseada nestes componentes discretos onde os subconjuntos de seus *traces* respeitam os requisitos discretos do sistema, respeitando os comportamentos da arquitetura (ambiente). O comportamento do controlador é definido a partir de comportamentos discretos das ações do ambiente e das reações esperadas do componente, uma vez que este é um controlador reativo.

Após a síntese do controlador, este é integrado a uma plataforma com componentes em Simulink (BEUCHER, 2006). Após esta integração, requisitos híbridos do sistema e comportamentos híbridos do ambiente são então validados durante a simulação destes componentes.

Esta metodologia utilizou uma ferramenta de síntese chamada *TuLiP* (WONGPIROM-

SARN et al., 2011). *TuLiP* é uma ferramenta de síntese de software de controle embarcado com base em um expressivo conjunto de formulas LTL. O estudo de caso utilizado para a validação desta metodologia resultou em um controlador reativo contendo 113 estados com um tempo síntese de aproximadamente 2 segundos. Todas as falhas de requisitos (satisfabilidade) e de comportamento de ambiente (compatibilidade) foram identificadas.

Este trabalho, apesar de ser uma metodologia de desenvolvimento para CPS, mais especificamente componentes de controle reativos, é de extrema importância para este estudo, uma vez que mostra a tendência e a viabilidade de projetos baseados em contratos para o desenvolvimento e integração de componentes de sistemas embarcados.

É importante lembrar que a abordagem de Nuzzo et al. (2013) descreve em primeiro momento os requisitos *top-level* do sistema. Em um projeto de sistemas, tanto requisitos quanto as propriedades de um ambiente são refinadas à medida que a níveis de abstração diminui Sangiovanni-Vincentelli, Damm e Passerone (2012), Gajski et al. (2009). Assim é extremamente importante ter modelos especializados, que acompanhem os refinamentos de contratos e permitam a validação da composição de componentes.

3.1.2 Ferrante 2014

Ferrante et al. (2014a) propõe uma abordagem baseada em contrato para especificar hierarquicamente interações entre diferentes componentes de um sistema, através de uma linguagem gráfica e textual chamada *Block-based Contract Language* (Block-based Contract Language (BCL)).

A objetivo desta abordagem consiste em lidar com problemas relacionados a definições de requisitos e suas validações no contexto de um sistema distribuído através de simulações de seus modelos Simulink (BEUCHER, 2006). A técnica proposta por (FERRANTE et al., 2014a) é fundamentada no clássico problema dos sistemas compostos por componentes projetados e desenvolvidos por diferentes empresas. Assim são especificadas as propriedades que ditam como o uso da interface destes componentes deve acontecer (contratos).

Como estudo de caso foi usado um módulo de controle de velocidade de cruzeiro de um automóvel, mostrado na Figura 12. Diversos sinais de sensores relacionados a velocidade e pedal de aceleração e de freio, por exemplo, são especificados como entrada deste módulo.

Inicialmente o projetista define informalmente os requisitos do sistema para em seguida formaliza-los através da linguagem BCL. O autor especificou no seu exemplo nove requisitos variando entre clausulas de suposições e garantias. Dois destes requisitos foram escolhidos para exemplificar o uso da abordagem:

1. A abertura da injeção de combustível implica em um movimento do carro (suposição).
2. Um comando de freio desabilita o controle de cruzeiro dentro de 3 segundos (garantia)

É importante deixar bem claro que os elementos destes dois requisitos são entradas do sistema apresentados na Figura 12: injetor de combustível (*THROTTLE*, no módulo de

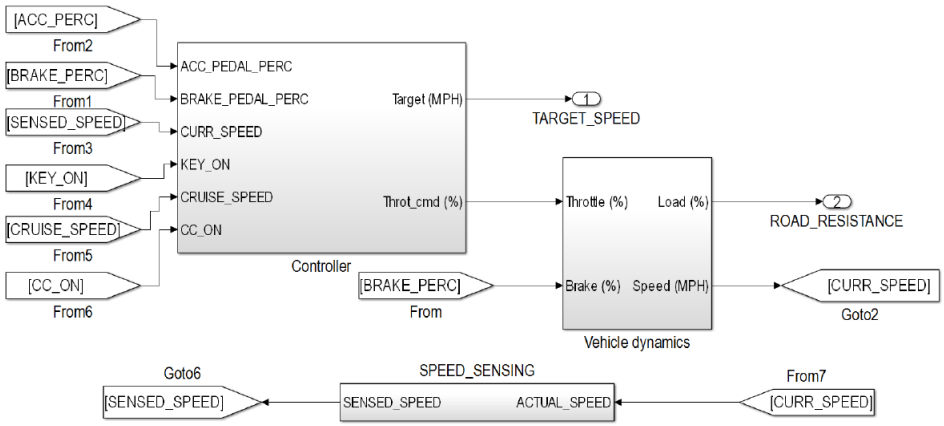


Figura 12 – Estudo de caso utilizado na validação da abordagem de Ferrante. Fonte: (FERRANTE et al., 2014a)

Vehicle Dynamics), sensor de velocidade (*SENSED_SPEED*, nos módulos *Controller* e *SPEED_SENSING*) e sensor de percentual de frenagem (*BRAKE_PERC*, nos módulos *Controller* e *Vehicle Dynamics*).

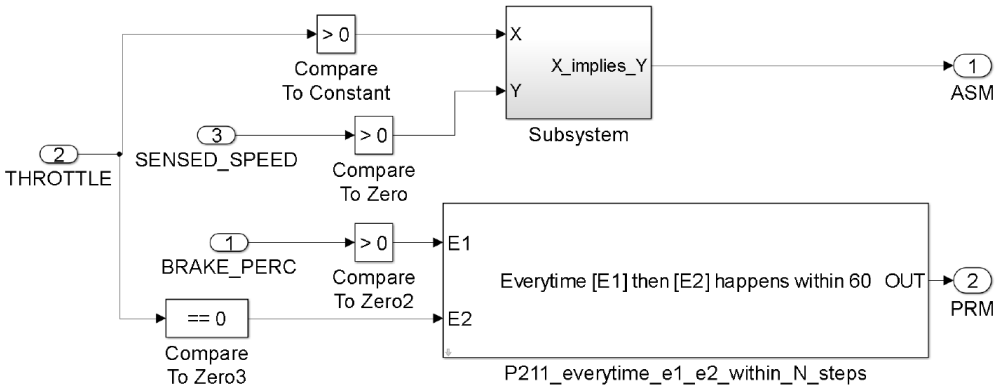


Figura 13 – Contrato de um requisito do estudo de caso proposto por Ferrante. Fonte: (FERRANTE et al., 2014a)

Com estes dois requisitos definidos, deve-se então formaliza-los através da linguagem BCL. A Figura 13 os mostra o resultado da formalização. As saídas deste diagrama, chamadas de *1(ASM)* e *2(PRM)* são os resultados lógicos das duas especificações, respectivamente. Pode-se perceber neste exemplo que, para as suposições, apenas blocos de comparação e um subsistema (*X_implies_Y*) foram utilizados para resolver estas comparações lógicas. Percebe-se que o uso de blocos ou funções predefinidas para construir os requisitos simplifica e clarifica bastante a definição da propriedade.

Porém, para construir a garantia, foi preciso escrever em LTL a propriedade de condição de tempo associada a um subsistema. Mesmo com este trabalho adicional, a linguagem é extremamente limpa e direta, reforçando a ideia de ser ter modelos bem definidos e amigáveis. Mesmo sem ler a definição informal dos requisitos é possível entendê-los

apenas olhando os diagramas. Para que o Simulink (BEUCHER, 2006) suporte este tipo de construção, foi preciso construir um conjunto de ferramentas (*plugin*) no próprio ambiente.

Dessa maneira, com os modelos definidos e suportados pela ferramenta, basta apenas executá-los e verificar se alguma suposição do ambiente (compatibilidade) e garantia (satisfabilidade) são violadas.

Neste exemplo, inclusive, houve uma violação do requisito 1 e esta foi detectada na simulação. A violação se deu pelo fato de que a abertura da injeção de combustível não implica em um movimento imediato do carro. Na Figura 13 é possível ver que para o 1 (ASM) ser verdadeiro, o *THROTTLE* maior que zero implica em o *SENSED_SPEED* ser maior que zero. Porém, nos momentos iniciais dessa ação, o carro não se move. É preciso que o combustível aja no motor e isso leva um pequeno tempo.

Dessa maneira o requisito precisou ser remodelado para "A abertura da injeção de combustível implica em um movimento do carro em até 1 segundo", onde em nova simulação a violação não mais ocorreu.

Como mencioando anteriormente, a linguagem BCL é extremamente legível, direta e muito amigável. Entretanto ela é voltada para sinais em interfaces de componentes, o que não se aplica na ideia proposta por este trabalho. Apesar de a interface ser fundamental para a validação que esta tese propõe, as propriedades dos protocolos de alto nível que se deseja validar não dependem diretamente dos sinais das interfaces dos componentes.

Este protocolos de alto nível são compostos através destes sinais, porém, especificar propriedades para estes protocolos com base neste sinais recaem na necessidade de especificá-las através de expressões longas, complexas e obscuras.

Além disso, esta técnica é aplicada apenas em simulação de componentes, não podendo ser aplicada na fase de operação, uma vez que tem toda a sua operabilidade vinculada à ferramenta Simulink (BEUCHER, 2006).

3.2 Técnicas Baseadas em Validação em Tempo de Execução

As técnicas apresentas a seguir são fundamentas em validações de hardware ou software, em tempo de execução, com base em checagem de propriedades formais.

3.2.1 Coutinho 2006

Coutinho (2006), sugere o *DesignMonitor*: um *framework* para detecção automática de violações de propriedades de aplicações Java concorrentes, em tempo de execução. Sua ideia básica é especificar previamente as propriedades de concorrência da aplicação e, de maneira não intrusiva e sem modificação do código fonte, monitorá-las em tempo de execução.

A Figura 14 mostra as cinco principais etapas desta abordagem. A primeira é a especificação das propriedades. Toda propriedade é identificada, tem o seu comportamento

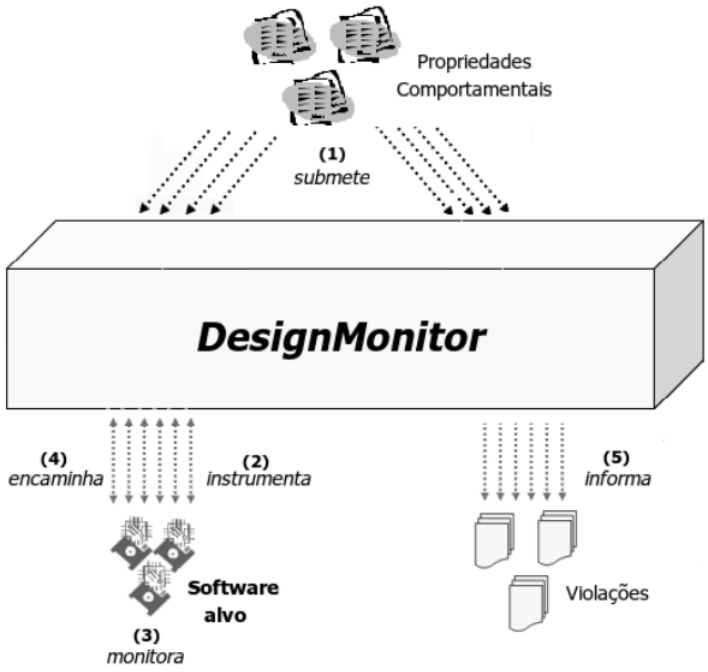


Figura 14 – Fluxo da abordagem proposta por Coutinho (2006). Fonte: (COUTINHO, 2006)

descrito através de formulas LTL e é associada a um ponto de interesse. Este ponto de interesse identifica o momento na execução no qual as propriedades são confrontadas.

Coutinho define as propriedades de uma determinada aplicação através de uma 3-tupla $S = \{P, C, R\}$ tal que P é o conjunto finito dos pontos de interesse, C é o conjunto finito de expressões do comportamento desejado e R é um conjunto da relação entre pontos de interesses e expressões de comportamentos, onde $R \subset \{PXC\}$.

Na segunda etapa é realizada a instrumentação da aplicação. Neste ponto não é preciso modificar ou anotar o código fonte da aplicação. Todos os pontos de interesse são especificados através de um formato compatível com AspectJ (KICZALES et al., 2001). Assim, ao invés de especificar a assinatura exata do método de uma classe referente ao ponto de interesse que irá disparar a checagem da propriedade, é especificado um padrão para a assinatura, semelhante a uma expressão regular.

Para que não haja um grande *overhead* na execução do framework, Coutinho sugere que o monitor esteja em uma máquina *host*, separada da máquina na qual a aplicação está sendo executada. Para isso Coutinho utiliza um outro *framework* chamado *Log4j*, envia o *log* da execução da aplicação para esta máquina remota (GROBMEIER, 2012).

A etapa três diz respeito ao monitoramento de fato. Todas as formulas LTL são transformadas em autômatos Büchi através da biblioteca *LTL2BA4J*, os quais são sensíveis a eventos gerados pelas chamadas realizados nos seus correspondentes pontos de interesse (GASTIN; ODDOUX, 2001). Assim, a cada evento, os comportamentos são confrontados com a execução da aplicação e, na etapa quatro, o resultado desta avaliação é encaminhada para o *DesignMonitor*.

Finalmente, na etapa cinco, caso haja alguma violação, o *DesignMonitor* informará qual das propriedades foi violada.

Um estudo de caso utilizando a implementação de um *middleware* para computação em grade (*grid*) foi usado para validar a abordagem. Este *middleware* é desenvolvido por uma equipe de programadores que compartilham o código por meio de um serviço de controle versão. Toda vez que os códigos são submetidos para o controle de versão, o *DesignMonitor* faz a sua validação. São 568 métodos de teste (identificados como pontos de interesse) e para monitorá-los o *DesignMonitor* gerou um *overhead* de aproximadamente 29%, o que é um valor não desprezível. Foram identificadas 190 violações de propriedades.

Esta abordagem, apesar de ser específica para aplicações concorrentes Java, deixa claro que o uso de asserções em monitoramento em tempo de execução é uma técnica viável e robusta. Entretanto, esta técnica ainda é dependente do código fonte da aplicação, uma vez que é preciso conhecimento de nomes e assinaturas dos métodos/serviços implementados. Para fins puramente de verificação de uma aplicação em desenvolvimento isto não um problema aparente, mas quando pretende levar em consideração um sistema completo composto por componentes de terceiros, esta abordagem encara o problema de nem sempre ter acesso ao código de deste outros componentes, o que limita a especificação e consequentemente o confronto das propriedades.

Apesar de não ser uma técnica para sistemas embarcados e consequentemente para HdS, o uso de asserção expressa em fórmula LTL reforça sua viabilidade inclusive para a aplicações em concorrência.

3.2.2 Pellizzoni 2008

A técnica proposta por Pellizzoni et al. (2008) consiste em um monitoramento de hardware em tempo de execução da comunicação entre *drivers* e componentes COTS (componentes pre-fabricados e vendidos em escala) em sistemas embarcados. Apesar de o próprio autor dizer que esta abordagem foi implementada e validada especificamente para dispositivos PCI, ele sugere que alguns princípios e conhecimentos aprendidos podem ser aplicadas para outros dispositivos.

De acordo com Pellizzoni et al. (2008), componentes COTS vêm tendo seu uso cada vez mais comum em sistemas embarcados. Entretanto, COTS, por não serem componentes customizados produzidos para um determinado e único projeto específico, quando integrados em uma plataforma alvo, têm sua imprevisibilidade como sendo uma séria ameaça para execução de sistemas críticos.

Com base neste problema é proposta uma técnica de monitoramento com a finalidade de verificar se certas propriedades críticas dos componentes estão corretas. Para isso, o trabalho propõe uma síntese automática do código fonte de monitores em hardware a partir de uma especificação formal em alto nível.

```
1   declarations : {
        signal cntrlCurrent : STD_LOGIC_VECTOR(15 downto 0) := X"0000";
3       signal cntrlOld : STD_LOGIC_VECTOR(15 downto 0) := X"0000";
    }
5   event countDisable : memory write address = base1 + X"220"
        dbyte value in "----------------0"
7   event cntrlMod : memory write address in base1 + X"220"
        {
9           cntrlOld <= cntrlCurrent; cntrlCurrent <= value(15 downto 0);
        }
11  event countEnable : memory write address = base1 + X"220"
        dbyte value in "----------------1"
13
    ere: ((countEnable countDisable) + cntrlMod + countDisable)*
15
    violation handler : {
17      mem_reg <= '1';
        address_reg <= base1 + X"220";
19      -- roll back to the previous cntr_cntrl2 value
        value_reg(15 downto 0) <= cntrlOld;
21      cntrlCurrent <= cntrlOld; enable_reg <= "0011";
    }
```

Figura 15 – Exemplo de uma propriedade especificada em *BusMOP* utilizada na abordagem de Pellizzoni et al. (2008). Fonte: (PELLIZZONI et al., 2008)

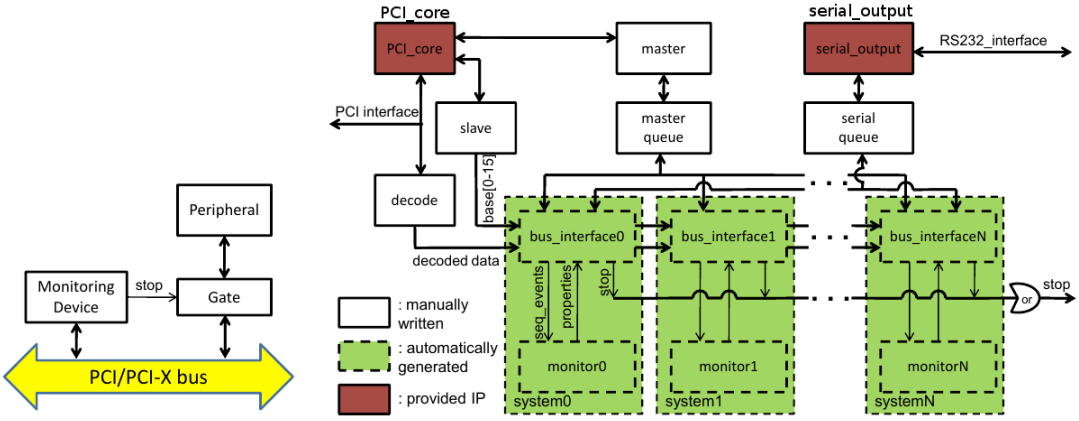
O trabalho é baseado no *framework Monitoring-Oriented Programming* (Monitor Oriented Programming (MOP)) (CHEN; ROSU, 2007). MOP é um *framework* formal de desenvolvimento e análise de software no qual o desenvolvedor especifica as propriedades desejadas através de formalismos específicos ou definidos por terceiros através de *plugins*, juntamente com trechos de códigos a serem executados quando há alguma violação ou validação de propriedades.

É importante deixar claro que este *framework* é basicamente aplicado em desenvolvimento de software. Por ser bastante expansível e suportar diversos formalismos, foi criada uma instância chamada *BusMOP*, voltado para sistemas embarcados (PELLIZZONI et al., 2008).

Abordagens MOP, como *JavaMOP* por exemplo, anotam o código a ser monitorado com uma especificação das propriedades além dos trechos de código ou funções que disparam os eventos que compõem estas propriedades.

O *BusMOP* tenta seguir a mesma ideia de especificação de propriedades, entretanto esta extensão do MOP não tem seus formalismos anotados em códigos fontes e seus eventos são disparados através de acessos realizados em um barramento. Entretanto, como dito anteriormente, esta técnica é específica para o barramento é o Peripheral Component Interconnect (PCI).

A Figura 15 mostra um exemplo de uma especificação *BusMOP* proposta nesta abordagem. A especificação é segmentada em quatro seções: declarações, eventos, propriedade e tratamento da violação. A seção de declarações, que pode ser vista na Figura 15, entre as linhas 1 e 4, são equivalente a declarações de sinais ou registradores na linguagem VHDL. Estes sinais ou registradores são utilizados para adicionar contexto aos monitores.



(a) Gated Monitoring Device. (b) Block Diagram.

Figura 16 – Arquitetura dos monitores propostos e do ambiente de integração da abordagem de Pellizzoni et al. (2008). Fonte: (PELLIZZONI et al., 2008)

O eventos são acontecimentos que ocorrem no barramento. Escritas e leituras em endereços com valores especificados podem desparar os eventos. Exemplos da declaração deste eventos podem ser vistos na Figura 15 entre as linhas 9 e 12

A propriedade de uma especificação *BusMOP*, cujo exemplo pode ser visto na Figura 15-linha 14, pode ser expressada através de uma fórmula LTL ou através de uma expressão regular estendida (Extended Regular Expression (ERE)).

Por fim, a seção de tratamento de violação, como pode ser vista na Figura 15 entre as linhas 16 e 22, permite a especifica um comportamento de reação ao se deparar com uma violação. Esta reação é escrita através da linguagem *VHDL*.

Assim, a partir de uma especificação *BusMOP*, monitores de propriedades são gerados como módulos de hardware. A Figura 16 mostra a arquitetura dos monitores e o ambiente no qual eles foram integrados.

Nesta abordagem existem componentes gerados automaticamente, componentes manualmente escritos e componentes fornecidos (IP). Estes componentes podem ser vistos na Figura 16 representados pelos blocos tracejados, blocos claros e blocos mais escuros, respectivamente. Os monitores são automaticamente gerados e são integrados ao barramento PCI como dispositivos mestres (*master*). Isso se faz necessário pelo fato de que os monitores podem interferir na configuração e uso dos periféricos no caso de uma validação ou violação de propriedade.

As interfaces mestre/escravo, assim como as filas de transações para estas interfaces, são manualmente implementadas. O módulo *decoder* funciona como um tradutor dos eventos que ocorrem no barramento, sendo o *front-end* para os *traces* de acessos a serem checados.

Dependendo da ocorrência de alguma violação, os monitores podem desativar o dispositivo através do sinal *stop*, disparado pelos módulos *bus_interface* dos monitores

gerados.

Apesar de a técnica validar a abordagem de maneira bem-sucedida, alguns fatores não viabilizam o uso desta abordagem para solucionar os problema mencionados no sec:intro. São eles:

1. Apesar de o autor citar no título do trabalho que esta técnica suporta a validação de sistemas embarcados de tempo real, o mesmo afirma que a abordagem não consegue dar suporte a expressões com características de tempo real. Como a especificação se baseia em MOP e esta foi inicialmente voltada para softwares, não há nenhum tipo de suporta a especificação de tempo.

2. A linguagem de descrição de propriedades não permite expressar propriedades pontuais com base em estados e protocolos, sendo todas elas globais. Isto recai no problema de se ter expressões longas e complexas quando há necessidade de especificar comportamentos mais detalhados ou especificos de sub-módulos. Porém isso ocorre somente para o *BusMOP*, uma vez que especificações como *JavaMOP* atribuem propriedades a classes ou métodos específicos. Como *BusMOP* não tem dependência com o código do *driver* e o *front-end* de seus eventos é a linha de barramento, não há especificação de escopo das propriedades dentro do protocolo.

3. Uma característica do BusMOP que torna a especificação um pouco distante da especificação informal dos dispositivos é o fato de que as variáveis com compõem os eventos (proposições) nas especificações MOP foram substituídas por endereços absolutos dos registradores dos periféricos, escritos em valores numéricos.

4. A abordagem dá suporte apenas ao monitoramento em plataforma real. Não é possível aplicá-la em todo ciclo de desenvolvimento do sistema, tanto devido aos monitores gerados serem especificados apenas em RTL quanto ao fato de a linguagem não ser dá suporte para a reformulação das propriedades de maneira incremental em refinamentos do modelo.

3.2.3 Zheng 2015

Zheng et al. (2015) apresenta um *framework* de especificação de propriedades de CPS e sua síntese em monitores de checagem em tempo de execução, chamado *BraceAssertion*. Com o objetivo de evitar formalismo rebuscados, Zheng adota sua especificação com base em *Behavior-Driven Development* (Behavior-Driven Development (BDD)). BDD é uma técnica de desenvolvimento criada para tentar tornar mais claros desenvolvimentos baseados em testes (Test-Driven Development (TDD)), definindo propriedades a serem testadas em um padrão (*template*) de expressões baseadas em linguagens naturais.

BDD é um artifício da engenharia de software e tem sua especificação inicial descrita a partir de uma "história"no formato mostrado a seguir (SOLÍS; WANG, 2011):

"**Tal como** [*tarefa*]
Eu quero uma [*característica*]
Então eu consigo [*benefício*]"

Dessa maneira os usuários podem identificar o que as tarefas devem fazer, e após a realização delas, quais os benefícios.

Para complementar, os cenários que ditam como o sistema deve implementar as suas *características* são descritos seguindo o seguinte *template*:

"**Dado** [*contexto*]
e [*mais algum contexto*]...
Quando [*evento*]
Então [*resultado*]
e [*mais alguns resultados*]..."

É justamente usando essa sintaxe de especificação de cenários, porém em inglês, que Zheng et al. (2015) expressa as propriedades. Porém ele precisou incrementar a sintaxe adicionando algumas características temporais. Assim, com base nos operadores LTL, Zheng adicionou as palavras *Always*, *Eventually*, *Before* e *After* para dar expressão temporal às suas propriedades.

A Figura 17 mostra o fluxo da abordagem proposta por Zheng. As propriedades são codificadas como métodos em classes Java e ao serem compiladas, monitores contendo consigo máquinas de estados que representando as formulas LTL são gerados.

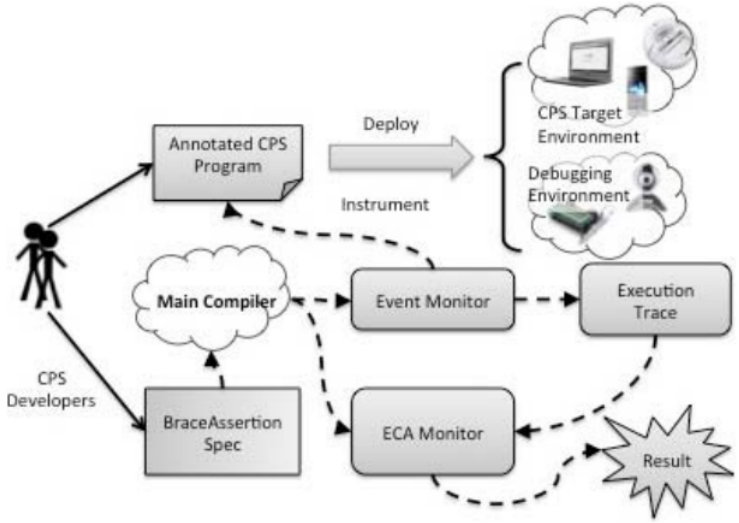


Figura 17 – Fluxo da abordagem proposta por Zheng et al. (2015). Fonte: (ZHENG et al., 2015)

Existem dois tipos de monitores: monitores de evento e monitores *Event Clock Automata*

(Event Clock Automata (ECA)). Os monitores de evento, com base em pontos de interesse AspectJ (KICZALES et al., 2001) anotados no software após sua compilação, montam um *trace* filtrado da execução do software. Já os monitores ECA, que são basicamente autômatos com informação de tempo, avaliam esse *trace* e indicam se as propriedades foram violadas.

Apesar da simplicidade e da clareza das expressões, esta abordagem está um pouco distante em se enquadrar no escopo almejado deste trabalho proposto pelos seguintes motivos:

1. Dependência do código fonte. Esta abordagem necessita do código fonte e dos pontos de interesse identificados e associados a seus monitores. Pelo mesmo motivo anteriormente já citado, esta dependência não é satisfatória no atual caso.

2. Com agentes (pontos de interesse) juntamente com o código (instrumentação), esta abordagem adiciona *overhead* na execução. Em um experimento com agentes robôs definindo rotas com base no algoritmo A^*[1], houve um *overhead* de 1,09% no uso de CPU e 1% no consumo de memória. Mesmo sendo um baixo impacto, como dito anteriormente, instrumentação de código pode alterar as características temporais do software e, para o caso deste trabalho, isto não é aceitável.

3. A Abordagem foi desenvolvida para aplicações Java, mesmo que se tentasse aplicá-la em outro escopo, seus monitores usam bibliotecas Java, o que torna mais dependente desta linguagem.

3.2.4 Técnicas Baseadas em Validação de Software Embarcado e HdS

Como um sub-tópico da seção de técnicas baseadas em validação em tempo de execução, estas próximas abordagens propões a validação de software embarcado ou HdS de maneira semiformal, ou seja, a partir de requisitos expressos através de propriedades formais, porém sendo validados em tempo de execução/simulação.

3.2.4.1 Lettnin 2009

Uma técnica de verificação semiformal de software embarcado com base em asserções foi proposta por Lettnin (2009), Lettnin et al. (2009). Uma vez que asserções são expressões baseadas em eventos temporais, elas são largamente utilizadas em verificação de hardware, porém para software, até então, não era aplicada devido falta de suporte para especificação de referência temporal.

Para suprir esta lacuna, Lettnin (2009) propôs um mecanismo de verificação do software a partir de sua simulação do mesmo em SystemC. A abordagem adaptou o

[1] Lê-se "A-Estrela

SystemC Temporal Checker SCTC (WEISS et al., 2006), que tem como proposta monitorar eventos temporais de modelos de hardware em *SystemC*, para suportar também eventos de temporização e tipos de dados complexos, além dos sinais de hardware.

É possível expressar as asserções através de fórmulas Finite Linear Temporal Logic (FLTL), uma extensão da LTL que permite expressar valores temporais, ou *Property Specification Language* (Property Specification Language (PSL)), formadas por proposições encapsuladas em classes C++ abstratas, chamada *Proposition*. Esta abordagem sintetiza estas expressões em autômatos de aceitação e rejeição (*AR-automata*) (RUF et al., 2001), os quais são utilizados pelo SCTC.

Esta abordagem sugere duas técnicas para efetuar a verificação. Na primeira é utilizado um modelo SystemC do microprocessador para que execute o software embarcado. Neste caso a verificação é realizada com precisão de tempo, uma vez que a cada ciclo de *clock* as propriedades são checadas. Apensar da vantagem da precisão de tempo, esta técnica é bastante custosa, uma vez que, além da execução do software simulado no modelo de processador, os *AR-automata* contem estados intermediários para cada ciclo informado nas fórmulas. Assim, se uma expressão informa que uma proposição deve ocorrer N ciclos depois de outra, existirão N estados intermediários representando este intervalo de tempo.

A Figura 18 mostra o fluxo desta primeira abordagem de verificação. É possível ver que no passo "(a)" da figura o projetista especifica a software (código C) e as propriedades que vão ser verificadas. Em seguida no passo "(b)" o código C é convertido em uma especificação no formato *3-Address code* onde em seguida, no passo "(c)" é instrumentado. Esta instrumentação anota as chamadas de função, identifica variáveis de entrada (variáveis que só são lidas) e atribui a elas o retorno de uma função de estímulo de *testbench*, e por fim adiciona *flags* de controle de inicialização do software e erros. A anotação das funções e *flags* servem para indicar ao SCTC as posições da execução do software.

Assim, no passo "(e)" o programa em C anotado e um *testbench* gerado automaticamente são compilados para o processador alvo (compilação cruzada) juntamente com as definições de aleatoriedade dos valores dos estímulos do *testbench*, descritos pelo projetista no passo "(d)". O *testbench* então irá atribuir estes valores às variáveis de entrada das funções de estímulos anotadas no passo "(c)".

No passo "(f)" o binário do software é gerado e este será executado no modelo do microprocessador. No passo "(g)" é feito um *dump* do binário para se ter acesso aos endereços em memória das variáveis do software. Dessa maneira, no passo "(h)" uma ferramenta *PROP2C* identifica as variáveis usadas nas propriedades e extrai seus endereços do *dump* do binário. Com isso, um módulo chamado de *ESW_monitor* é gerado e este encapsula o monitor SCTC, repassando-o o conteúdo dos endereços das variáveis extraídas de um modelo de memória durante a execução do software no microprocessador.

A segunda técnica proposta por Lettnin envolve a execução do software, porém sem precisão de tempo. Para isso, todo o software é convertido para um modelo *SystemC*.

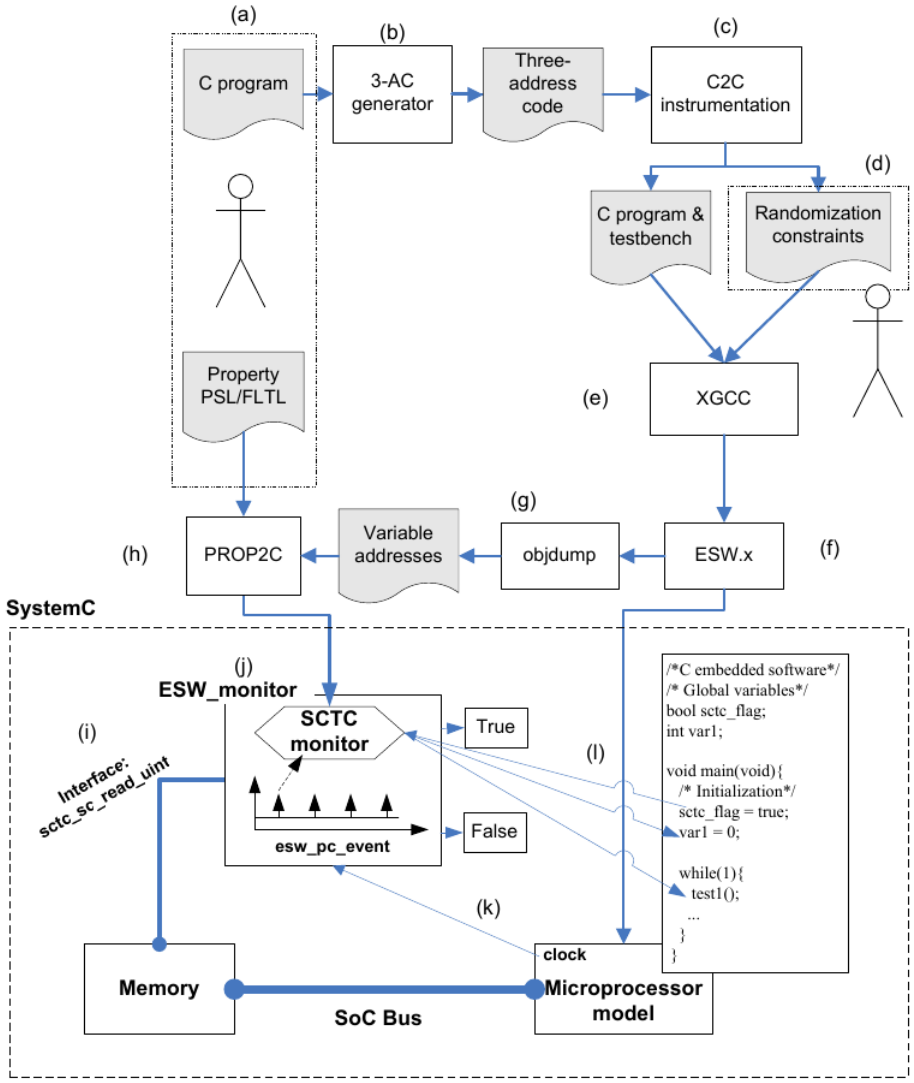


Figura 18 – Fluxo da abordagem com precisão de tempo proposta por Lettnin (2009). Fonte: (LETTNIN, 2009)

Dessa maneira ele não é simulado em um modelo de microprocessador, mas sim suas instruções são executadas nativamente, incluindo pontos de sincronização a cada declaração (*statement*) do programa, simulando contagem de tempo. Apesar da desvantagem de não contar com a precisão de ciclo do processador alvo, esta abordagem é muito mais rápida, uma vez que a execução nativa e o *AR-Autômato* gerado não necessitarão de um estado intermediário para cada ciclo de *clock*, mas sim a cada declaração do código C.

A Figura 19 mostra o fluxo desta segunda técnica. Uma diferença entre as técnicas está no passo "(c)", uma vez que não há apenas a instrumentação do código, mas sim a conversão do software para um modelo *SystemC* instrumentado com os pontos de sincronização e contagem de tempo, e anotação das chamadas de funções. Além disso, há também a conversão de toda atribuição e acessos a ponteiros (acesso direto à memória) em tabela hash, chamada de *VMEM*. Esta tabela dá o suporte a esta técnica para checagem de HdS, uma vez que a memória *VMEM* pode servir como entrada de valores do *testbench*,

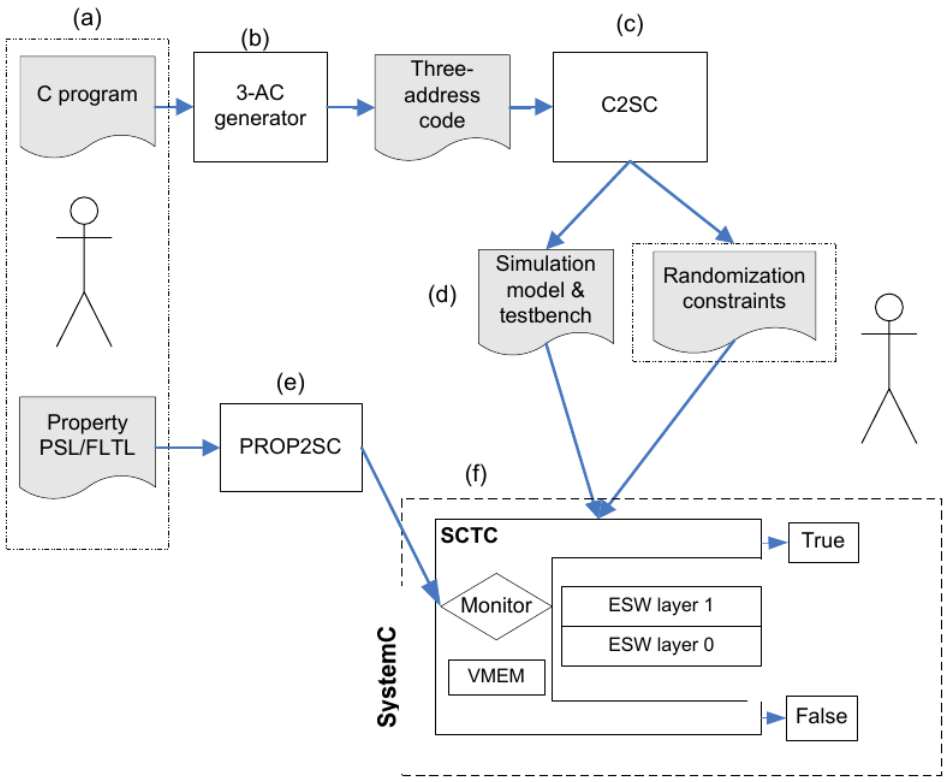


Figura 19 – Fluxo da abordagem sem precisão de tempo proposta por Lettnin (2009). Fonte: (LETTNIN, 2009)

simulando comportamentos de periféricos.

Apesar da eficiência em detectar previamente comportamentos indesejados em projetos de sistema, alguns pontos importantes devem ser observados em relação à sua aplicação em checagem de protocolo de comunicação entre *drivers* e periféricos:

1. Em relação à instrumentação de código fonte requisitada pela abordagem, em alguns casos ela pode diminuir o desempenho ou alterar o comportamento de temporização do protocolo. Assim, é interessante que as ferramentas ou técnicas utilizadas evitem modificar o código fonte do software. Vale salientar que, às vezes, por questões de metodologias de teste e verificação, elas nem sequer deveriam ter acesso a este código (REINBACHER et al., 2012).

2. O item anterior reforça um problema relacionado à dependência do código fonte, o que pode ser uma desvantagem para manutenção e detecções de alteração de comportamentos por intrusão. Qualquer modificação e atualização na camada de software pode implicar em uma modificação no ambiente de verificação. Nomes e endereços de variáveis no próprio código do projetista, bibliotecas e *frameworks* de terceiros e sistemas operacionais podem mudar de versões para versões. Essas modificações podem implicar em alterações nas propriedades especificadas. É importante que a checagem de protocolos de comunicação entre componentes seja orientada às suas

interfaces e ao protocolo de comunicação, sendo completamente independente de camadas do software.

3. Na primeira técnica proposta por Lettnin, como se faz uso do *dump* do binário para extrair os endereços das variáveis utilizadas nas expressões e há apenas um modelo de memória que guarda os valores, não há como simular o comportamento de periféricos, o que impossibilita a checagem de protocolos com comunicação com dispositivos. Dessa maneira não existe a possibilidade de efetuar uma verificação com precisão de tempo da comunicação entre *drivers* e dispositivos. Já a segunda técnica permite simular o comportamento dos periféricos, uma vez que a memória *VMEM* suporta a entrada de estímulos do *testbench*. Porém, nesta técnica não há precisão de tempo.

3.2.4.2 Reinbacher 2012

Reinbacher et al. (2012) propõe um ambiente de verificação em tempo de execução para software embarcado em microcontroladores baseado no monitoramento dos acessos às regiões mapeadas em memória. A Figura 20 mostra a arquitetura da abordagem proposta.

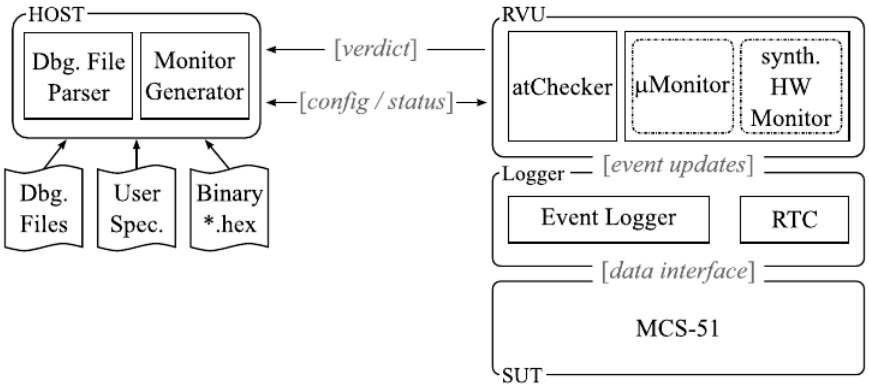


Figura 20 – Arquitetura da abordagem proposta por Reinbacher et al. (2012). Fonte: (REINBACHER et al., 2012)

Como pode ser visto, a arquitetura é dividida em dois ambientes físicos distintos: máquina hospedeira (*host*), à esquerda na Figura 20, e o ambiente de monitoramento, sintetizado em um FPGA, composto pelo *Runtime Verification Unit* (Runtime Verification Unit (RVU)), *Logger* e *System-Under-Test* (System-Under-Test (SUT)), à direita na Figura 20.

Toda a configuração do ambiente de monitoramento e visualização das propriedades violadas são feitas pela máquina hospedeira. Está última é conectada ao ambiente de monitoramento pelo RVU através de uma porta USB. Porém, antes de detalhar sobre o ambiente hospedeiro, é preciso detalhar cada componente do ambiente de monitoramento e explicar como a verificação é realizada.

O SUT, como o próprio nome já diz, é onde está sendo executado o sistema em teste. Nesta abordagem, Reinbacher et al. (2012) utilizou um microcontrolador *Intel MCS-51* (HUANG, 2000) executando o software sob verificação, compilado com a ferramenta *Keil* (ARM..., 2016).

As portas de entrada e saída do microcontrolador são monitoradas pela unidade *Logger*. Esta unidade é responsável por capturar os eventos que representam as mudanças nos endereços de memória e, através de um Real Time Clock (RTC), cada evento é registrado com um *log* de tempo. Assim, cada evento é enviado para RVU para que as propriedades sejam checadas.

A RVU é composta por dois tipos de componentes: os *atCheckers*, que são basicamente resolvedores de proposições atômicas, e um resolvedor de expressões LTL. Os resolvedores de LTL podem ser implementados como uma espécie de processador de LTL, chamado de μMonitor, ou as fórmulas LTL são sintetizadas diretamente no FPGA como máquinas de estados. É importante ressaltar que as propriedades são definidas com base nos valores Booleanos resolvidos pelos *atCheckers* durante a execução do sistema.

Já do lado do ambiente hospedeiro, existem dois componentes: o parser dos arquivos de depuração e o gerador de monitores. Os arquivos de depuração servem para informar os endereços das variáveis utilizadas, similarmente ao *dump* feito na abordagem de Lettnin (2009). Assim, com os arquivos de depuração juntamente com as especificações de propriedades do usuário, o gerador de monitor configura a memória do μMonitor com as expressões LTL, ou as sintetiza direitamente no FPGA. Assim o hospedeiro envia o binário do software para a memória do microcontrolador e então o ambiente está pronto para iniciar a execução e verificação.

Vale destacar que as propriedades especificadas pelo usuário são feitas de maneira literal, onde estas propriedades fazem referência às variáveis do software, entretanto de maneira global, sem nenhuma referência a estados ou etapas de protocolos. Reinbacher et al. (2012) chama a atenção para este fato e comenta que, se houver a necessidade de definir pontos de referência na execução do software, pode-se fazer uso do valor do *program counter* (PC).

Esta abordagem reforça a viabilidade de checagens em tempo de execução monitorando portas de entrada e saída. Um ponto positivo desta abordagem está no fato não necessitar instrumentação e ser completamente não intrusiva. Entretanto, como a abordagem de Lettnin (2009), ela também depende de informações do código fonte e depende do código binário, uma vez que suas expressões são vinculadas às variáveis do código, e necessita dos arquivos de depuração gerados na compilação para extrair os endereços das variáveis. Além disso apenas variáveis globais são checadas. Para esta abordagem proposta, este tipo de dependência não é interessante por motivos já comentando anteriormente, como por exemplo, a inutilização da validação no caso da modificação da implementação de software de terceiros.

Outra desvantagem para a aplicação desta abordagem para validação da comunicação entre periféricos é o fato que os eventos são disparados apenas nas mudanças de valores das variáveis, e não pelo acesso a elas. Isso pode ser um problema quando é necessário ter o conhecimento de uma leitura realizada, por exemplo. Isso acontece bastante quando um *driver* precisa ter conhecimento se o seu dispositivo está pronto antes de enviar algum comando, por exemplo.

3.2.4.3 Decker 2018

Decker et al. (2018) propõe uma técnica de monitoramento *online* da execução de software em processadores *multicores*.

Processadores *multicores* modernos contam, para cada um dos seus *cores*, com uma unidade de observação da execução do software, conhecida como Embedded Trace Unit (ETU), a qual disponibiliza para o meio externo um *trace* compactado desta execução e do estado do processador.

Dependendo da sua implementação, uma ETU pode fornecer informações como, por exemplo, endereços de instruções executadas pelos núcleos da Central Processing Unit (CPU), escalonamento de tarefas e gatilhos de exceções e ocorrências de operações que acessam registradores, memória e periféricos.

Entretanto, até o momento, os dados emitidos pelas ETU são tratados e estes são avaliados em de maneira *offline*. Mas isto acarreta é uma quantidade muito grande de dados e tratá-los consome muito tempo e requisita um pode computacional elevado.

Por esse motivo, Decker et al. (2018) propõe a avaliação *online* destes dados, mas para isso é preciso construir um monitor que, além de receber os dados, deve também conhecer a semântica dessas informações para poder interpretá-las, e pra isto este trabalho propõe a síntese de um ambiente de monitoramento contando com monitores para cada *core*, extraídos a partir de uma linguagem de alto nível, chamada de *Tessla*.

Porém, a abordagem não se resume a uma especificação na linguagem *Tessla*. Há duas etapas *offline* que precisam ser realizadas, sendo uma antes de outra após a análise *online*. A Figura 21 mostra o fluxo da abordagem e a arquitetura do ambiente sintetizado com seus monitores.

A primeira etapa, chamada de *Configuration*, é a etapa de pré-processamento. Nesta etapa o binário do software que será executado na plataforma é desmontado e o Control-Flow Graph (CFG). A partir deste CFG, um Waypoint Graph (WPG) é calculado, e pra isso um resolvedor de padrões verifica se cada instrução é uma instrução de *waypoint*. Esses instruções são especificadas no manual do *ARM CoreSight Program Flow Trace*.

Com os CFG e WPG, é possível especificar um modelo através da linguagem *Tessla* vinculando pontos da execução destes grafos. A Figura 22 mostra um exemplo de uma especificação utilizando esta linguagem.

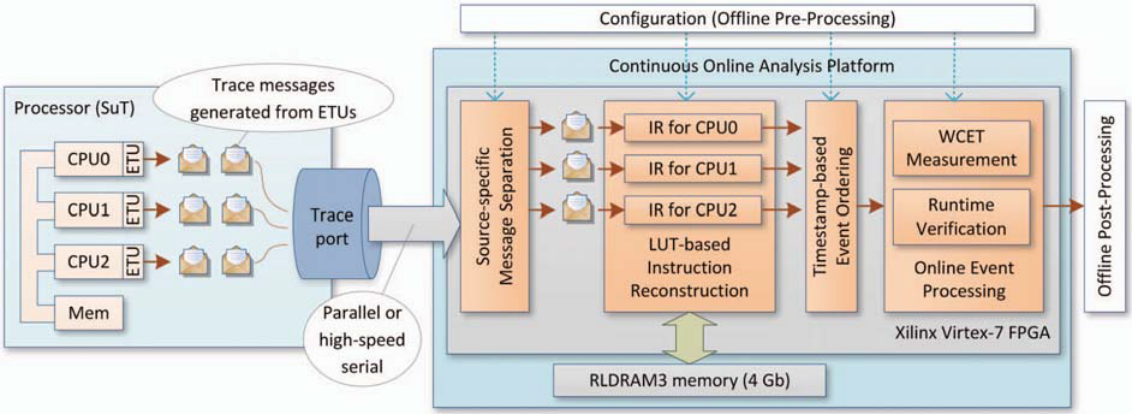


Figura 21 – Arquitetura da abordagem proposta por Decker et al. (2018). Fonte: (DECKER et al., 2018)

```
1 define block_start := onExecuting 0x2000
2 define block_end := onExecuting 0x101c
3 define timing_min_violation := on block_end if inPast(block_start, 16us)
4 define timing_max_violation := on block_end if not inPast(block_start, 23us)
5 out timing_min_violation
6 out timing_max_violation
```

Figura 22 – Exemplo da descrição da interface de comunicação na linguagem TDevC

É possível visualizar através desta Figura que a linguagem permita especificar pontos nos no código fonte, associados aos *waypoints* (linhas 1 e 2), e é possível definir relações de ordem e temporais entre estes *waypoints* (linhas 3 e 4). Estas relações são formulas LTL descritas em uma sintaxe mais próxima da linguagem natural. Na especificação também é possível informar dados que serão repassados para a última etapa do fluxo da abordagem, pós processamento, através da palavra reservada *out* (linhas 5 e 6).

Após o programa executado, a fase de pós-processamento é iniciada. Nesta fase é possível realizar o download da informação montada (estatísticas de tempo, estatísticas de cobertura de código, relatórios da verificação de tempo de execução e tempo-real). Esses informações são disponibilizadas ao usuário e podem ser utilizadas para outros processamentos futuros.

Como mencionado no inicio desta seção, este trabalho permite identificar momento de acessos a periféricos. Assim, é importante vincular o contexto deste trabalho ao proposto nesta tese.

As propriedades especificadas pelo usuário nesta abordagem também são feitas de maneira literal, e mais uma vez as propriedades fazem referência às pontos do software e também de maneira global, sem nenhuma maneira de fazer referência a estados ou etapas do protocolo de comunicação com controladores periféricos.

Apesar do ponto positivo de ser aplicada a processadores *multicores*, o trabalho não menciona a possibilidade de especificar pontos de controle de seção crítica para a acessos concorrentes a periféricos.

Um outro ponto positivo desta abordagem está no fato não necessitar instrumentação e ser completamente não intrusiva. Entretanto, como a abordagem de Lettnin (2009) e Reinbacher et al. (2012), ela também depende de informações do código fonte e depende do código binário, uma vez que necessita do binárioa para extrair os CFG e WPG e as expressões especificadas através da linguagem *Tessla* são vinculadas à pontos do código

Mais uma vez, para esta abordagem proposta, este tipo de dependência não é interessante por motivos já comentando anteriormente, como por exemplo, a inutilização da validação no caso da modificação da implementação de software de terceiros.

Outra grande vantagem desta abordagem se dá na possibilidade de especificar propriedades com relação de tempo discreto. Esta característica é fundamental para a identificação de erros causados por violações de tempo-real.

3.2.4.4 Macieira 2011

Um sistema de monitoramento de propriedades temporais descritas em modelos de alto nível de abstração através de monitores sintetizados a partir destas descrições e integrados em plataformas virtuais foi proposto por Macieira, Lisboa e Barros (2011). Esta técnica surgiu da ideia de se validar os *device drivers* gerados a partir da abordagem proposta por Lisboa et al. (2009).

Lisboa et al. (2009) propõe uma linguagem de domínio específico (DSL), chamada *DevC*, para a descrição em alto nível de características de dispositivos. A partir desta descrição do dispositivo, um modelo SystemC do controlador do dispositivo e uma versão simplificada do seu *device driver* são sintetizados usando uma ferramenta de síntese chamada dcsim.

Como mencionado no início deste capítulo, técnicas de geração de *drivers* corretos por construção não sintetizam estes componentes completamente. Assim, ainda se faz necessária a codificação manual.

Justamente por este motivo, a técnica proposta por Macieira, Lisboa e Barros (2011) agregou ao fluxo da técnica proposta por Lisboa et al. (2009) a etapa de checagem. Como pode ser visto na Figura 23, as etapas sinalizadas como *(a)* e *(b)* são relativas à geração do modelo SystemC do controlador e do *driver* simplificado, respectivamente. Estas etapas fazem parte da abordagem proposta por Lisboa. Já as etapas *(c)* e *(d)* foram propostas por Macieira e são as etapas relacionadas à checagem das propriedades durante a simulação.

Para consolidar a fusão das duas técnicas, foi proposta uma extensão da linguagem *DevC*, nomeada de Temporal DevC, uma vez que esta passou a suportar a descrição de características temporais relacionadas a comportamentos dos *device drivers* (MACIEIRA; LISBOA; BARROS, 2011; MACIEIRA; BARROS; ASCENDINA, 2014). Estes comportamentos são expressados através da descrição de sequências de acessos a registradores analisados sempre que é realizada uma chamada de função do *driver*.

Os monitores sintetizados são integrados na plataforma virtual e ficam monitorando toda a comunicação entra a CPU e os dispositivos da plataforma, inclusive a memória.

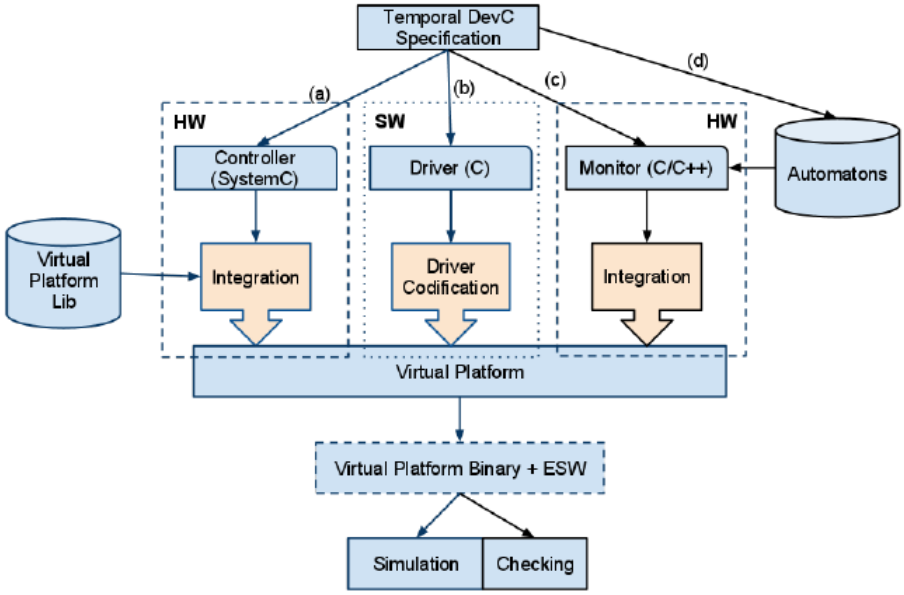


Figura 23 – Fluxo da abordagem proposta por Macieira, Lisboa e Barros (2011). Fonte: (MACIEIRA; LISBOA; BARROS, 2011)

Assim, para que o monitor saiba quando uma chamada de função foi realizada, é preciso que ele tenha conhecimento do seu endereço em memória.

Devido a esta necessidade, a síntese destes monitores exige tanto o conhecimento dos nomes das funções, uma vez que eles são associados na descrição Temporal DevC, quanto dos seus endereços. Logo, o binário do software embarcado também é essencial.

Apesar desta técnica não ser intrusiva em relação à execução da plataforma embarcada e permitir a descrição de propriedades com escopos definidos nas funções do *driver*, a dependência do código fonte e do binário do software é uma desvantagem para o seu uso além da fase de desenvolvimento do sistema, uma vez que isto a torna totalmente sensível a alterações na pilha de software do sistema.

Complementando, a linguagem proposta por Macieira, Lisboa e Barros (2011) também não permite a descrição de propriedades com informação de tempo, além de que o poder de expressão fornecido pela Temporal DevC é extremamente limitado se comparado ao poder de expressão de formulas LTL ou PSL, por exemplo. A sintaxe da linguagem é bastante restrita e só permite especificar comportamentos simplificados, como sequencias limitadas de acessos as registradores, e estes comportamentos somente são checadas se houver a chamada da função previamente definida.

3.3 Análise Comparativa

A análise comparativa dos trabalhos relacionados pode ser vista na Tabela 1, na qual as vantagens e desvantagens de cada abordagem em relação aos objetivos propostos por este trabalho são mostrados. As características avaliadas nesta comparação foram:

1. **Suporte a HdS**: este parâmetro avalia se a abordagem permite checar propriedades além do escopo do software. Se Permite checar também, além acessos a endereços de memória, acessos a outros periféricos de uma plataforma.

2. **Suporte a fase de desenvolvimento**: este parâmetro avalia se a abordagem dá suporte durante a fase de desenvolvimento do *device driver*.

3. **Suporte ao desenvolvimento incremental**: este parâmetro avalia se a abordagem suporta especificar as propriedades de maneira incremental, sem modificação ou substituição de propriedades.

4. **Suporte durante fase de operação do sistema**: este parâmetro avalia se a abordagem pode ser aplicada durante a fase de operação do sistema, independente de componentes externo à plataforma.

5. **Suporte a propriedades definidas em modos de operação ou estados**: este parâmetro avalia se a abordagem suporta relacionar propriedades temporais especificadas a escopos da execução, a modos de operação ou a estados.

6. **Suporte a restrições temporais**: este parâmetro avalia se a abordagem permite especificar e validar propriedades com restrições temporais com base na especificação de tempo discreto. Propriedades temporais são fundamentadas em eventos os quais não necessariamente são valores temporais discretos, mas sim a ocorrência de algum acontecimento (evento). As formulas temporais que descrevem estas propriedades normalmente não vinculam valores discretos ou o quantitativos de eventos para definir a veracidade da lógica, mas sim contam com operadores temporais que relacionam a ocorrência destes eventos. Porém, para alguns casos, como eventos de tempo-real, é interessante ter o quantitativo discreto da ocorrência deste eventos.

7. **Independência do binário do software embarcado**: este parâmetro avalia se a abordagem não precisa ter acesso ou depende de informações do binário do software embarcado, ou se pode ser influenciada com alguma mudança no software.

8. **Independência do código fonte do software embarcado**: este parâmetro considera se a abordagem não precisa ter acesso e depender de informações do código fonte do software embarcado, ou ser influenciada com a sua mudança

9. **Insensibilidade a bibliotecas de software de terceiros**: este parâmetro avalia se a abordagem não é influenciada ou inviabilizada com a existência ou mudança de biblioteca de terceiros

10. **Execução da checagem de maneira não intrusiva**: este parâmetro considera se a abordagem não afeta a execução do sistema, ou seja, se ela é totalmente transparente

para o software em execução, não afetando assim seu tempo de execução e consumo de memória, mantendo exatamente fieis suas características de tempo de execução.

11. **Não necessita fazer anotações no código do software**: este parâmetro considera se a abordagem necessita realizar alguma anotação ou modificação no código fonte do software para que esta possa realizar a checagem das propriedades.

12. **Validação da consistência das propriedades**: este parâmetro considera se a abordagem valida as propriedades, garantindo que não há inconsistências nas suas descrições.

13. **Suporte a Concorrência**: este parâmetro considera se a abordagem dá algum suporte à validação de propriedades relacionadas a concorrência no acesso de recursos e seções críticas.

Na Tabela 1, tab:works, associa cada abordagem apresentada com estas métricos em questão.

Apenas as três abordagens relacionadas a geração de *drivers* corretos por construção tem as características (5), (7) e (10) classificadas dessa maneira. Isso se dá pelo fato de que para estas abordagens não há especificação de propriedades a serem checadas (5), o binário é somente gerado após a aplicação da abordagem (7) e não há intrusão, uma vez que não há monitoramento e o *driver* é o único componente gerado e integrado pela abordagem.

Das abordagens que realizam validação ou verificação em tempo de execução que suportam HdS, quando permitem a definição de propriedades em escopos da execução ou de maneira incremental, acabam dependendo do binário ou código fonte do software embarcado, que servem como guias para as propriedades. Assim, estas abordagens são orientadas às implementações do software. Consequentemente elas acabam sendo sensíveis a bibliotecas de softwares de terceiros. Isto ocorre pelo fato de que, como há uma dependência de informações do código fonte ou binário do software, variações em bibliotecas de terceiros podem influenciar ou mascarar os endereçamentos das variáveis relacionadas às propriedades.

Nas abordagens de Lettnin et al. (2009) e Zheng e Julien (2015), há ainda a necessidade de anotação do código fonte, tornando maior a dependência, uma vez que não basta apenas conhecer os nomes das variáveis e funções do software, mas sim alterá-lo para que este faça chamadas de funções ou modificações em memória que avisem ao monitor de propriedades que algo importante aconteceu ou seu contexto se encontra em algum ponto de checagem de propriedades durante a simulação ou execução.

É importante relembrar que anotar ou modificar o código significa modificar suas propriedades temporais. Mudanças na alocação de memória e novas chamadas de funções adicionam *overhead* na execução e podem acarretar em violações de restrições de temporais.

É fundamental falar também sobre a intrusão na execução do sistema. Todas a abordagens apresentadas que não dependem de código fonte, não realizam a validação em tempo de execução de maneira intrusiva. Isso se dá pelo fato de que os seus mecanismos de checagem não são guiados pelo software, mas sim por uma especificação do componente. No caso da técnica proposta por Pellizzoni et al. (2008) o suporte é parcial e se dá pelo fato de sua abordagem só ser intrusiva quando seus monitores tomam ações corretivas diante de uma violação. Logo, é possível usá-la de maneira não intrusiva.

É possível perceber que quando a abordagem é orientada à implementação do software embarcado, mesmo independente de suporte a HdS, o inverso quase sempre é verdade, com exceção da técnica proposta por Reinbacher et al. (2012) e Decker et al. (2018). Isso acontece pelo fato de que quando se tem noção dos endereços de variáveis locais ou funções e de suas aplicações no contexto do software, é possível definir propriedades dentro dos escopos desses elementos, mesmo que de maneira parcial. Na abordagem proposta por Reinbacher et al. (2012) seria possível de delimitar escopos, mas em sua abordagem ele optou por definir propriedades apenas com base em variáveis globais.

Para a abordagem proposta nesta tese, essa dependência de código fonte é evitada. Como o objetivo é checar a corretude no uso de periféricos por parte do *driver*, independente das camadas do software embarcado, uma validação orientada à especificação possibilita forcar puramente no protocolo, independentemente de como este vai ser ou é implementado. Além disso, permite a separação da equipe responsável por validar ou verificar o sistema da equipe que desenvolveu o código fonte e de vícios da implementação.

Outra característica observada foi a possibilidade de aplicar a técnica em conjunto com metodologias incrementais de desenvolvimento como a baseada em plataforma. Apesar de algumas abordagens estarem presentes tanto na fase de desenvolvimento quanto na fase de operação, quando isso acontece elas suportam apenas o desenvolvimento em hardware real, não podendo ser aplicada em modelos em alto nível de abstração, em uma fase inicial do projeto, com o propósito de serem analisados e refinados incrementalmente. Porém, quando a abordagem suporta o uso em modelos de alto nível e suporta simulação, ela pode ser aplicada a várias fase de desenvolvimento.

Um parâmetro observado na comparação foi o suporte à validação prévia das propriedades especificadas. Nenhuma das abordagens avaliadas faz uma validação de consistência das propriedades. Este tipo de validação previne a descrição de propriedades contraditórias e consequentemente evita a realização de uma validação ou verificação errada desnecessária. Mesmo que propriedades contraditórias sejam identificadas durante o monitoramento pelo fato de sempre haver uma violação em curso, quando existem escopos ou contextos da execução associados às propriedades, essas violações por contradição podem demorar para aparecer. Assim, dependendo do tempo de projeto alocado para a checagem do sistema, a existência de propriedades contraditórias pode ser bastante prejudicial, uma vez que a inconsistência da especificação pode somente ser observada depois grande tempo de

execução.

A abordagem proposta por Nuzzo et al. (2013) não efetua de fato uma validação prévia, mas como esta abordagem usa as propriedades para efetuar a síntese de controladores reativos, se houver alguma contradição entre as propriedades não haverá pelo menos um *trace* válido para se extrair um possível controlador. Porém, como esta abordagem faz uso de propriedades para a geração do controlador e propriedades para a sua simulação, pode haver algum caso em que a identificação das propriedades contraditórias usadas unicamente para a simulação seja também feita tardiamente.

Por fim, já em relação ao suporte à validação de acessos concorrentes a recursos e seções críticas, nenhuma abordagem dá um suporte especializado a este tipo de propriedades. Três das abordagens analisadas dão um suporte parcial, uma vez que duas instrumentam o código, adicionando informações para as propriedades em relação a alguma concorrência existente, ou, como no caso da abordagem proposta por Ferrante et al. (2014b), a análise de propriedades é feita em nível de sinais. Dessa maneira, é possível identificar a concorrência através dos próprios sinais especificando uma propriedade que leve em consideração esta concorrência.

Com base nesta análise, percebe-se que se faz necessária uma abordagem que suporte o desenvolvimento de *device drivers* robustos e que possa ser usada em conjunto com as metodologias atuais de projetos de sistemas. É fundamental também que esta abordagem auxilie na segurança de uso dos componentes mesmo após o seu desenvolvimento, dando suporte durante a fase de operação do sistema. Para isso, a abordagem precisa validar propriedades de protocolos de comunicação como sequências e permissões de acessos e restrições temporais, além de não poder ser dependente de implementações ou códigos de terceiros, uma vez que em um sistema computacional a pilha de software quase sempre pode sofrer modificações benignas, como em atualizações, ou modificações malignas, no caso de invasões por *malware*.

Tabela 1 – Comparação entre os trabalhos relacionados

Caract. / Abord.	Suporta						Indeped.		(9) Insensib. a libs de SW de terceiros	(10) Exec. não Intr.	(11) Sem Anot de Cód.	(12) Val. das Propr.	(13) Concorrência
	(1) HdS	(2) Fase Desenv.	(3) Desenv. Increm.	(4) Fase Oper.	(5) Prop. Modo Oper/ Estado	(6) Restr. temp.	(7) Bin.	(8) Cód. Fonte					
Cout. '06	×	■	×	□	■	×	×	×	×	×	■	×	×
Pelizz. '08	■	■	×	■	□	×	■	■	■	□	■	×	×
Lettnin'09	□	■	×	×	□	□	□	×	×	×	×	×	□
Macieira'11	■	■	×	×	□	×	×	×	×	■	■	×	×
Reinb. '12	×	■	×	×	×	■	×	×	×	■	■	×	×
Nuzzo'14	■	■	■	×	×	■	■	■	■	■	■	□	×
Ferrante'14	■	■	■	×	■	■	■	■	■	■	■	×	□
Zheng'15	×	■	×	■	■	■	×	×	×	×	×	×	□
Decker '18	■	■	×	□	×	■	×	×	×	■	■	×	□

Suporta	■
S. Parcial	□
N. Suporta	×

3.4 Resumo

Este capítulo apresentou uma série de trabalhos que atacam problemas relacionados à validação de sistemas e que serviram como base de estudo para a concepção da abordagem em proposta. No fim do capítulo foi feita uma análise comparativa entre os principais trabalhos e esta análise destacou quais pontos ainda não são tratados pelas abordagens existentes, servindo como motivação para esta tese.

4 VISÃO GERAL DA ABORDAGEM PROPOSTA

Como mencionado no Capítulo 1 desta tese, para realizar o monitoramento da comunicação entre *drivers* e dispositivos, esta abordagem propõe uma técnica que inclui uma linguagem para a descrição de restrições a serem monitoradas, bem como a arquitetura de um módulo de monitoramento sintetizado a partir das descrições destas restrições. Os monitores de propriedades temporais (as quais representam as restrições) chamados *monitors of driver/device communication* (MDDC), ficam observando o meio de comunicação entre a CPU e os dispositivos durante a execução de uma plataforma embarcada e verificam se as restrições estão sendo respeitadas no intuito de que a plataforma execute de maneira correta e confiável.

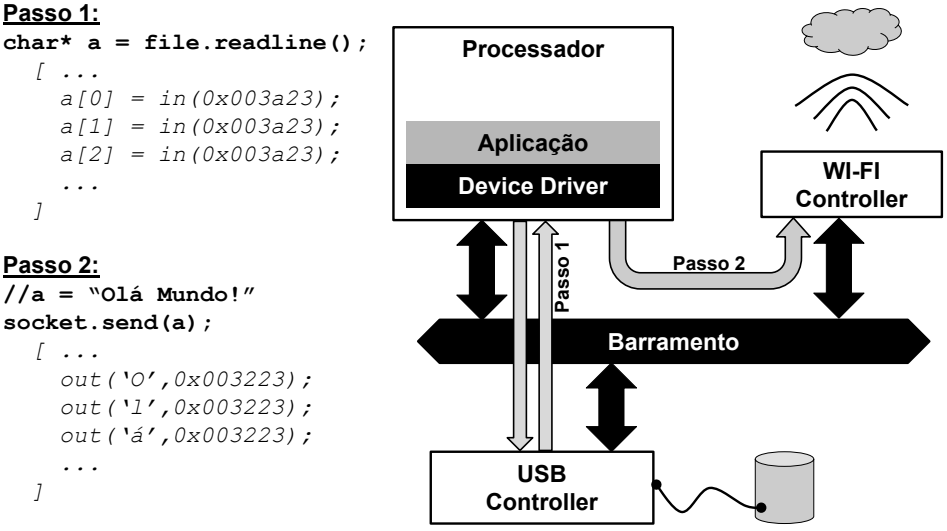


Figura 24 – Exemplo simplificado da atuação de um *driver* em uma plataforma embarcada

A Figura 24 demostra um exemplo simplificado do funcionamento de uma plataforma embarcada contendo dois dispositivos: um controlador WI-FI e um controlador USB. Neste exemplo, a aplicação sendo executada no processador deseja ler uma mensagem de texto armazenada em um dispositivo USB e deseja também enviar esta mensagem para um servidor remoto via rede WI-FI. Para isto, a aplicação deve invocar funções dos *device drivers* dos periféricos para que estes façam tanto a leitura do texto no dispositivo USB quando o envio do texto via rede *wireless*.

Basicamente, o que estas funções dos respectivos *device drivers* fazem é traduzir estas requisições em sequencias de acessos aos endereços dos respectivos dispositivos, como foi explicado no Capítulo 1. Estas sequencias na Figura 24 podem ser vistas tanto no pseudo-código no lado esquerdo da figura, quanto no diagrama da plataforma, no lado direito da figura. No *Passo 1* a CPU requisita sequencialmente os dados do texto ao dispositivo USB através de leituras no endereço 0x003a23 e no Passo 2 a CPU solicita o

envio dos dados do texto, de maneira sequencial, ao controlador WI-FI, através de escritas no endereço 0x003223.

Porém, se algum destes *device drivers* for implementado de uma certa maneira em que algum passo dessas sequências seja ignorado ou que algum acesso seja feito no momento ou endereço indevido, este dispositivo pode realizar uma ação inesperada pelo *driver* ou, o que é mais crítico, levar o dispositivo para um estado indefinido.

Por este motivo, este trabalho propõe o monitoramento desta comunicação entre CPU e periféricos, com o intuito de identificar possíveis erros de implementação dos *drivers*, o que implica no uso errado do protocolo de comunicação especificado pelo fabricante do dispositivo.

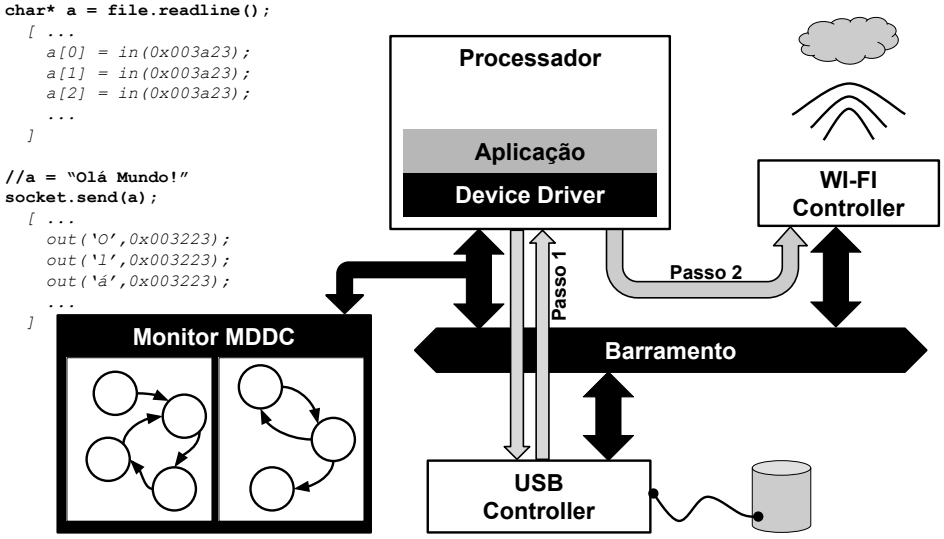


Figura 25 – Exemplo simplificado da atuação do MDDC

A Figura 25 mostra o mesmo exemplo mostrado na Figura 24, porém com o monitor MDDC integrado à plataforma. O MDDC contém modelos de referência representados através de máquinas de estados e, como o MDDC observa a comunicação entre o processador e os periféricos, em cada etapa da sequencia de acesso a um determinado dispositivo, o monitor pode efetuar transições de estados nessas máquinas de estados.

Para isso o monitor faz uso de dois modelos de máquinas de estados: a máquina de estados finita hierárquica com dados, chamada de *FSM-Monitor* e o autômato Büchi (Büchi Automaton (BA)), os quais terão os seus papeis na abordagem descritos mais adiante.

Um MDDC pode monitorar a comunicação entre a CPU e diversos dispositivos em uma única plataforma. Assim, a instância do monitor contém um conjunto distinto de informações sobre cada dispositivo sob validação, sendo a principal informação de cada um deles o seu protocolo básico de comunicação. Este protocolo é descrito através da *FSM-Monitor*, que reflete exatamente a execução do protocolo de comunicação esperado pelo dispositivo, servindo como um modelo de referência. Do ponto de vista de contratos,

esta máquina de estados reflete as **suposições** em relação ao uso do protocolo por parte dos *device drivers*.

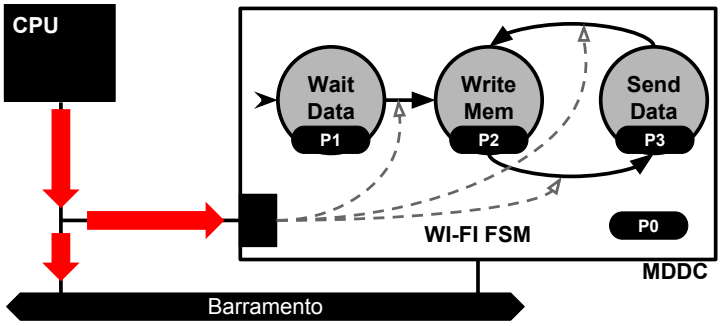


Figura 26 – Exemplo simplificado do funcionamento interno do MDDC

A Figura 26 mostra um exemplo simplificado de uma *FSM-Monitor* representando o modelo de referência do exemplo do controlador *WI-FI* apresentado na Figura 25, bem como a maneira como ela reflete o fluxo do protocolo durante a comunicação com o dispositivo sob validação e demonstra como uma escrita a este dispositivo modifica o estado do monitor.

Assim que o acesso ao registrador é realizado e o MDDC o aceita como válido, o acesso é traduzido em transições da *FSM-Monitor*. Com base no exemplo da Figura 26, o estado inicial da máquina de estados é o estado *WaitData*, indicando que o dispositivo está aguardando a escrita de algum dado a ser enviado. Assim, como a *FSM-Monitor* é um modelo de referência do protocolo de comunicação entre o *driver* e o dispositivo sob monitoramento, espera-se que o dispositivo seja inicializado em um estado *WaitData* internamente (suposição).

Quando o monitor recebe como entrada mais um acesso válido, dependendo do acesso, este pode resultar em uma transição na *FSM-Monitor* como, no caso da Figura 26, do estado *WaitData* para o estado *WriteMem*, caso o *driver* tenha solicitado o a escrita de algum dado na memoria do periférico. É importante chamar a atenção para o fato de que para a transição de *WaitData* para *WriteMem* ocorrer é preciso que a CPU tenha solicitado esta escrita e que o MDDC esteja preparado e ciente de que essa solicitação é o acesso que disparará essa transição na *FSM-Monitor* do estado *WaitData* para o estado *WriteMem* (suposição).

Olhando mais uma vez a Figura 26, pode-se perceber que cada estado contém um bloco representando o conjunto das propriedades que aquele estado deve respeitar. São eles: *P1* para o estado *WaitData*, *P2* para o estado *WriteMem* e P3 para o estado *SendData*. Existe também um bloco de propriedades (*P0*) fora de qualquer estado. Isso quer dizer que as propriedades pertencentes ao conjunto *P0* são globais. Nas seções seguintes os escopos e hierarquias tanto das propriedades quanto dos estados serão explicados com mais detalhes.

Trazendo mais uma vez para o ponto de vista de contratos, as propriedades, são as garantias agregadas ao fluxo de execução do protocolo (estados) com base nas suposições

especificadas. Assim, as transições de estados formam os *traces* básicos que definem os protocolos e as propriedades são incorporadas a estes *traces* agregando restrições tanto de satisfabilidade (quando são relacionadas às garantias) quanto de compatibilidade (quando são relacionadas às suposições).

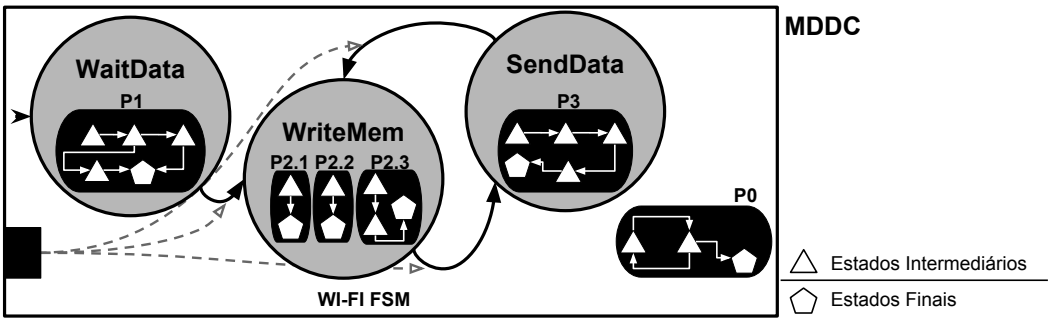


Figura 27 – Ilustração simplificada dos estados da FSM-Monitor com o seu conjunto de propriedades

Essas propriedades, cujos estados devem respeitar, são implementadas através das máquinas de estados. A Figura 27 mostra cada estado com o seu conjunto de propriedades ou restrições, onde apenas *P2* contém mais de uma propriedade, somando 3 no seu total.

Os autômatos Büchi (GASTIN; ODDOUX, 2001) são as representações formais de propriedades temporais, ou seja, propriedades que devem ser respeitadas ao longo do tempo. Com base em eventos temporais [1], elas efetuam as transições de estados sempre após a atualização dos estados e valores da *FSM-Monitor*. Nesta abordagem, os eventos temporais são os acessos da CPU aos dispositivos, interrupções e tempo de relógio quando são envolvidas propriedades de tempo real.

Na Figura 27 os triângulos representam ou os estados iniciais ou os intermediários e os pentágonos os estados finais ou de aceitação. Quando um estado de aceitação é alcançado implica dizer que a propriedade foi violada, uma vez que as restrições são especificas. Em outras palavras, especificar uma restrição significa especificar quando esta restrição ocorre. Por este motivo, alcançar o estado final, significa identificar uma violação de restrição.

4.1 Fluxo da Abordagem

Implementar as máquinas de estados e integrar os MDDC utilizando uma linguagem de propósito geral ou uma Hardware Description Language (HDL) não são tarefas simples e podem consumir uma boa parcela de tempo de projeto, uma vez que cada MDDC é específico para cada plataforma alvo.

Além disso, a implementação completa de um mecanismo de monitoramento utilizando linguagens de propósito geral o torna bastante suscetível a erros, uma vez que a codificação

[1] Eventos que ocorrem em intervalos de tempo, não necessariamente em tempos constantes. Exemplo de eventos temporais: *Clock* em circuitos integrados, transações em sistemas bancários, envio e recebimento de pacotes de rede, **acesso a dispositivos** e etc.

manual das tarefas funcionais e não funcionais do projeto precisa ser levada em conta. Assim, é ideal que um ambiente de monitoramento seja especificado em um alto nível de abstração e levando-se apenas em consideração as propriedades na comunicação com os dispositivos.

Desta maneira, esta abordagem propõe um mecanismo para suportar o desenvolvimento de um ambiente de monitoramento através de uma sequência de passos que pode ser vista no fluxo mostrado na Figura 28

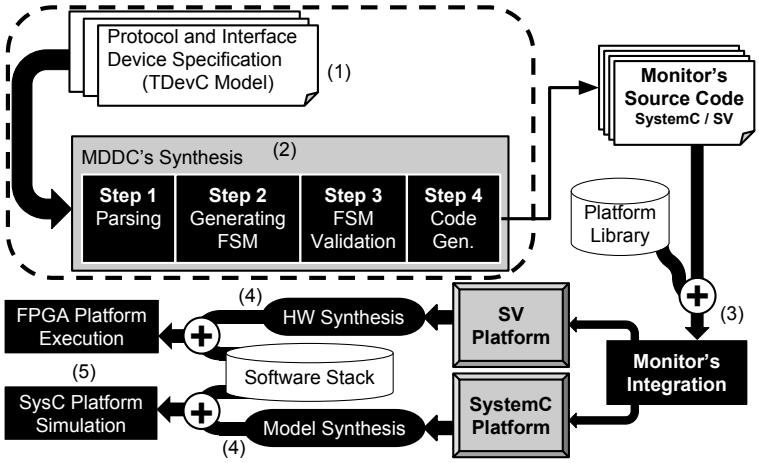


Figura 28 – Fluxo da Abordagem

Este fluxo tem como início a especificação, em alto nível de abstração, do protocolo de comunicação entre o processador e o dispositivo e das restrições de uso deste protocolo, cujo *driver* está sob validação. Esta especificação é realizada através da linguagem proposta denominada *TDevC*.

A *TDevC* suporta em sua sintaxe construções próprias para a especificação de estruturas como registradores, seus formatos e seus campos, além de padrões e máscaras de dados. Ela também contém construções para a especificação de comunicação com dispositivos, tais como protocolos de acesso aos registradores, bem como máquina de estados refletindo o comportamento do dispositivo e propriedades temporais representando comportamentos específicos que eventualmente venham a ocorrer durande a execução do sistema. Mais detalhes sobre a linguagem *TDevC* serão apresentados na Capítulo 6.

Após a especificação de um modelo em *TDevC*, uma ferramenta chamada *TDevCGen* realiza a síntese do monitor MDDC a partir deste modelo. Dividida em 4 passos, a síntese tem início com o *parsing* da especificação em TDevC. A *TDevCGen* então cria as listas de controle dos elementos de interface e de protocolo descritos na linguagem e em seguida, no passo 2, transforma o modelo descrito em *TDevC* em um formato intermediário, montando em memória uma máquina de estados hierárquica e vinculando aos seus estados os elementos de interface associados às suas transições.

Com o formato intermediário armazenado em memória, a *TDevCGen* então efetua o passo 3 que consiste na validação do modelo especificado. Esta validação inclui a verificação

de inconsistências e não-determinismos na máquina de estados bem como a verificação se existem contradições entre as propriedades comportamentais de estados e seus descendentes hierárquicos. A Figura 29 mostra a interface gráfica da ferramenta *TDevCGen* após o *parsing* de uma especificação *TDevC*. É possível ver informações sobre registradores, partes da *FSM-Monitor* e de propriedades no formato de máquinas de estados.

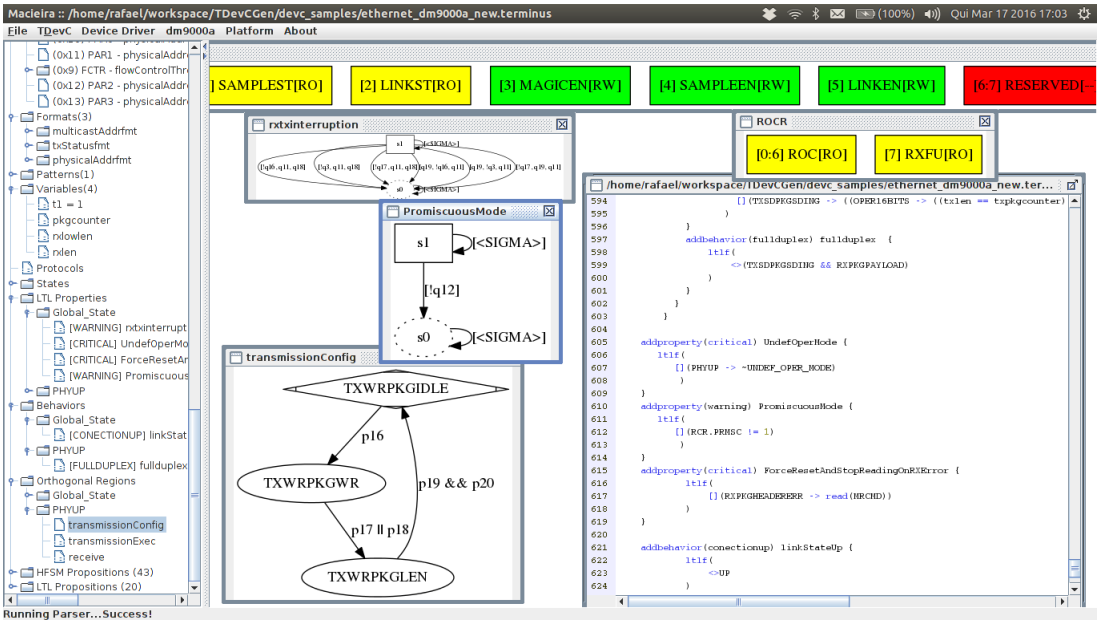


Figura 29 – Interface gráfica da TDevCGen

Uma vez que o modelo foi validado, segue-se para o passo 4, no qual a ferramenta então gera o monitor em C++ ou em SystemVerilog sintetizável. O componente monitor é então manualmente integrado na plataforma.

O passo 4 é realizado por um módulo da ferramenta *TDevCGen* no qual, atualmente, está configurado apenas para sintetizar o MDDC nestas duas linguagens. Entretanto, como se trata de uma ferramenta altamente modularizada, é perfeitamente possível e simples adicionar um modulo de geração de código fonte para outra linguagem que seja desejada. Há uma iniciativa inclusive de transformar este módulo em um canal de integração externo para que o usuário possa adicionar modelos de geradores de código fontes com base no modelo de *plugins*.

A etapa de integração consiste em conectar o MDDC gerado na plataforma alvo. Uma vez que as portas físicas (baixo-nível de abstração) ou funções (alto-nível de abstração) associadas aos dispositivos mestres interceptam os acessos a endereços de memória, os dados dos acessos são enviados para a porta de observação do MDDC. No monitor, estes dados são analisados e comparados com as transições nas máquinas de estados. Os detalhes desta etapa serão melhor explicados na sec:integração.

Após a integração manual, o MDDC está pronto para interceptar todos os dados enviados a partir de um dispositivo mestre para um determinado periférico durante a

execução de um software embarcado e determinar se estes acessos representam uma violação ou não do protocolo de comunicação.

Todos os dados trocados entre o dispositivo mestre e o periférico sob validação são analisados e, dependendo do tipo de acesso, do endereço relacionado e do valor lido ou atribuído, uma transição de estado na máquina hierárquica do MDDC pode ser disparada. Assim, com a máquina de estados hierárquica refletindo o comportamento do dispositivo e associando as propriedades comportamentais aos estados, o monitor pode detectar, em qualquer momento da execução do sistema, o status do periférico e a ocorrência dos comportamentos descritos associados ao estado corrente.

É importante destacar que, além do monitor ser um mecanismo de espionagem usado para observar a comunicação entre elementos mestres e escravos da plataforma, ele também pode ser considerado como um dispositivo escravo (periférico). Logo, a CPU pode configurá-lo e, na ocorrência de um determinado comportamento especificado, requisitá-lo para obter mais informações sobre o evento. Como dito anteriormente, esta característica é interessante, não só para a detecção de comportamentos indesejados e depuração de software embarcado, mas como também para a tomada de decisão diante de ocorrências de falhas.

4.2 Arquitetura do MDDC

Se a descrição TDevC da *FSM-Monitor* é validada com sucesso, em seguida a ferramenta gera o módulo MDDC e ele já pode ser integrado na plataforma sob monitoramento. Atualmente, esta etapa de integração é realizada manualmente.

Após a integração na plataforma, o MDDC está pronto para observar todos os dados enviados a partir da CPU para um determinado periférico para determinar se o acesso é válido ou não. Com base na máquina de estados que reflete o comportamento esperado, todos os dados trocados entre a CPU e o dispositivo são avaliados e, dependendo do tipo de acesso, do endereço acessado e do valor lido ou atribuído, ocorre uma transição na respectiva *FSM-Monitor*.

Como mostrado na Figura 30, o módulo MDDC é composto por cinco subunidades: *Bus Snooping Interface (Bus Snooping Interface (BSPI))*, *Protocol Translator (Protocol Translator (PT))*, *Bus Slave Interface (Bus Slave Interface (BSI))*, porta externa e o *FSM Controller*, contendo o *FSM-Set*.

O sub-módulo BSPI é responsável por capturar todos os pacotes de dados trocados entre o processador e os dispositivos e, através do *Device Selector (Device Selector (DS))*, seleciona apenas os dados relacionados com os dispositivos cujos *device drivers* estão sob monitoramento. Isso pode ser facilmente feito verificando a faixa dos endereços dos acessos capturados. Se o endereço acessado está na faixa de endereço de qualquer dispositivo sob monitoramento, o BSPI ativa sua respectiva máquina de estados. Em seguida, todos os dados relacionados ao acesso são transferidos para a respectiva *FSM-Monitor* na FSM-Set,

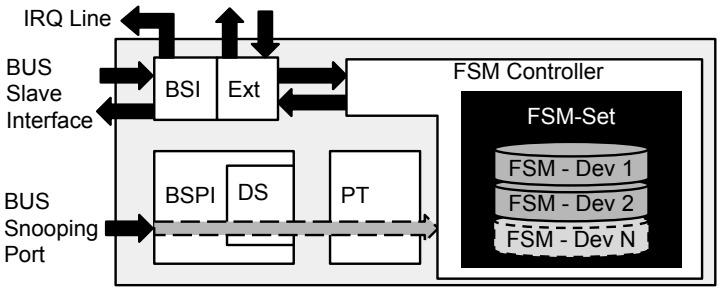


Figura 30 – Arquitetura do MDDC

onde serão processados. É importante notar também que a geração do BSPI depende da plataforma. Assim, esta unidade varia de acordo com a interface que está sendo utilizada entre a CPU e os periféricos e é integrada ao MDDC com base em uma biblioteca de componentes de interface.

Esta biblioteca deve ser previamente implementada pelo projetista. Atualmente esta implementação é feita de maneira manual, apesar de esta abordagem já esta sendo estendida para suportar a implementação desta biblioteca através da própria especificação TDevC. Entretanto, é importante chamar a atenção para o fato de que os módulos BSPI desenvolvidos até o momento contém apenas algumas dezenas de linhas de código, não ultrapassando 100 linhas de código.

Outra questão bastante importante de se destacar é o fato de que, com o BSPI definido, esta abordagem suporta qualquer modelo de processador, uma vez que este esteja integrado no barramento. O MDDC é completamente independente de conjuntos de instruções ou modelos de processadores, uma vez que as informações fundamentais para seu funcionamento são baseadas em endereço de registrador, tipo de acesso (leitura ou escrita) e os valores lidos ou escritos. Reafirmando o que já foi discutido, este monitor é totalmente independente da pilha de software em execução, sendo totalmente transparente para o monitor em qual conjunto de instruções o software foi compilado.

Apesar desta futura extensão da TDevC, é bom reforçar a importância de ter estas duas especificações separadas. Protocolos de comunicação em alto nível de abstração são diferentes e independentes do meio o qual os dados desta comunicação são trafegados. Logo, independentemente do meio de transmissão utilizado, o protocolo de comunicação de alto nível sempre se manterá o mesmo, sendo complemente independentes do meio nos quais os componentes estão conectados.

A BSI é uma porta escrava de barramento (*bus slave port*). Ele conecta o MDDC à plataforma como um dispositivo comum. Esta porta é usada para a configuração do MDDC e para obter informações das propriedades quando estas são violadas.

O módulo *Protocol Translator* (PT), como o nome já explica, é responsável por traduzir o protocolo de acesso às memórias internas ou registradores do dispositivo sob monitoramento. Em alguns casos os endereços variam numa faixa muita restrita na plataforma ou os dispositivos fornecem apenas alguns poucos registradores externos

(mapeados com endereços de barramento). Assim, para acessar registradores internos, um protocolo deve ser utilizado ao longo destes endereços e registros externos. Estes protocolos são especificados na TDevC através das construções *protocol* e sintetizados neste módulo.

O FSM-Set, como já explicado, é o conjunto de todas as *FSM-Monitor* dos dispositivos sob monitoramento. O módulo BSPI, através do módulo PT, notifica o módulo FSM-Set sobre cada acesso feito a um dispositivo. No caminho inverso, o módulo FSM-Set notifica o módulo BSI sobre a ocorrência de um comportamento não esperado. Através de um pedido de interrupção (Interrupt Request (IRQ)), o módulo BSI notifica *driver* do MDDC executando no processador sobre a ocorrência de qualquer comportamento. Logo, o *driver* pode solicitar de volta para o MDDC as informações sobre esse comportamento (qual dispositivo, tipo de comportamento, tipo de acesso, estado, valor lido ou escrito e endereço).

Como também já comentado anteriormente, o uso de interrupção ou acesso realizado pela própria plataforma sob monitoramento é opcional. Através da porta externa (*Ext*, na Figura 30), é possível que um sistema remoto controle e receba todas as informações sobre o monitoramento. Desta maneira o monitoramento se torna 100% transparente e não intrusivo, não provocando depreciação do desempenho do sistema.

Em relação à latência do monitoramento, da entrada do sinal até a transição de estados dentro da FSM-Set, são consumidos 3 ciclos de relógio. Um ciclo para a tradução do protocolo no BSPI e para a detecção do dispositivo no DS, outro ciclo consumido na tradução de mapeamento de registradores internos no PT e outro ciclo na transição de estados nas FSM-Monitors.

É importante destacar que essa latência sempre se mantém, uma vez que todo evento que dispara uma atividade no monitor sempre será um evento de acesso a registradores, o que gera um comportamento totalmente previsível nos módulos do MDDC. A única questão imprevisível é para qual estado ocorrerá uma transição, mas independente do estado alvo, sempre será um único passo por por vez, mesmo que haja mais de uma transição no mesmo evento. Em outros palavras, para cada região ortogonal, sempre ocorrerá apenas uma única transição por evento. Isto garante a previsibilidade de comportamento do MDDC.

Apesar da latência de 3 ciclos, todos os módulos funcionam como estágios de um pipeline. Assim, logo que o primeiro módulo entrega a informação para o módulo seguinte (do BSPI + DS para o PT, por exemplo), este já está pronto para receber novos dados. Dessa maneiras, o MDDC está pronto para receber a entrada de eventos a cada ciclo de relógio.

Outra informação fundamental é que o MDDC, mais especificamente cada FSM-Monitor, permite o registro de *log* do *trace* mais recente, ou seja, dos últimos estados percorridos. Para a implementação em SystemVerilog, foi definido um *log* que registra sempre apenas os últimos 8 estados percorridos por região ortogonal, por uma questão de restrição de recursos. Já para a implementação em *SystemC*, não foi definido limites, uma vez que os recursos são mais abundantes.

4.3 Abordagem Proposta e Metodologias de Desenvolvimento

Conhecendo em detalhes a abordagem proposta, é possível agora entender como esta técnica se enquadra nas metodologias de desenvolvimento apresentadas no Capítulo 2.

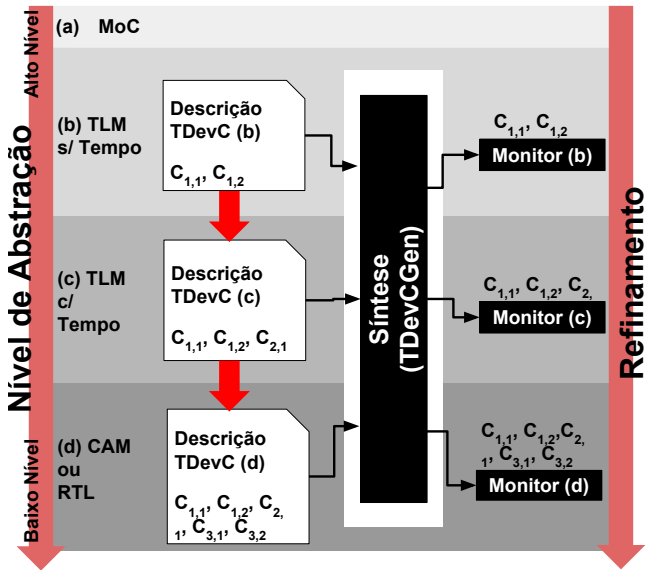


Figura 31 – Processo incremental da abordagem proposta

Como mostra a Figura 31, a cada refinamento e incremento da plataforma é possível realizar o refinamento e incremento dos modelos *TDevC* simplesmente adicionando mais detalhes, sem refazer ou modificar a especificação prévia. Como a *TDevC* suporta uma especificação hierárquica, é possível adicionar detalhes aos elementos já previamente especificados, agregando novas cláusulas nos contratos à medida que mais detalhes vão sendo incorporados na plataforma.

Assim, através da *TDevCGen*, em cada iteração de refinamento extrai-se um monitor a partir da especificação *TDevC*. Estes monitores agregam as novas cláusulas dos contratos e já podem ser integrados na plataforma nas diferentes fase do desenvolvimento, como pode ser visto na Figura 32.

A integração do monitor na plataforma é feita através de uma ligação entre a conexão do elemento de processamento e o meio de transmissão compartilhado de dados. Esta sonda permite a interceptação não intrusiva e a observação da comunicação do processador com os outros dispositivos da plataforma. Colhendo informações dessa comunicação, o monitor checa se as cláusulas dos contratos especificados sob o protocolo de comunicação estão sendo respeitadas.

O monitor da última iteração pode ser gerado também em RTL e sintetizado. Assim, ele pode ser integrado permanentemente na plataforma alvo e pode estar conectado a sistemas julgadores externos, informando-os sobre qualquer anomalia na comunicação de periféricos ou funcionar como um sensor virtual de propriedades para sistemas com base nas asserções especificadas no modelo *TDevC*. Estes sensores virtuais funcionam como

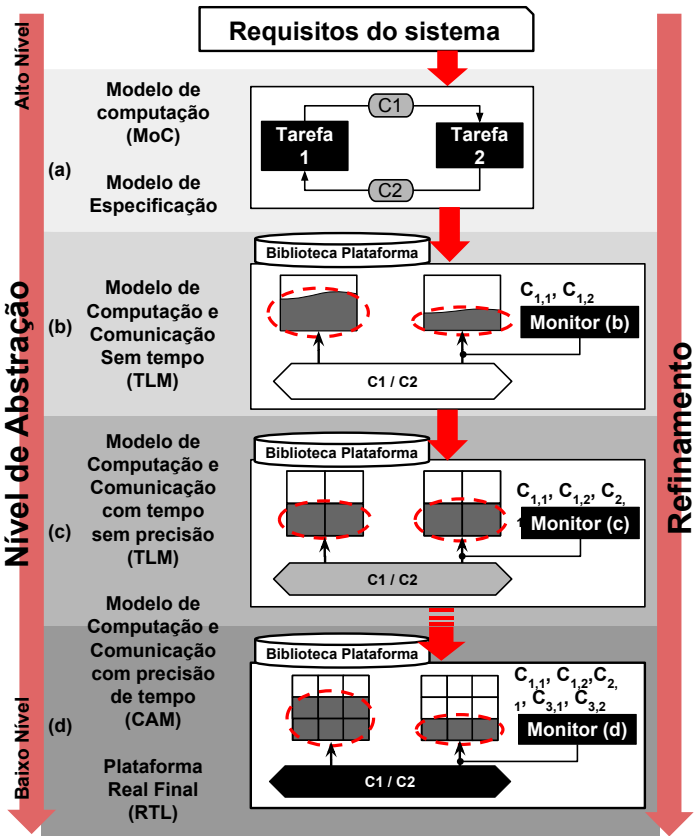


Figura 32 – Integração da abordagem em projetos de sistemas

sensores lógicos detectando comportamentos anômalos na comunicação com os dispositivos e os informando à CPU, onde esta adapta a plataforma quando necessário como um *feedback loop* virtual (SARMA et al., 2014; SARMA et al., 2015).

Atualmente estão sendo implementados os monitores para plataforma virtual com base nos monitores criados previamente por Macieira, Lisboa e Barros (2011).

4.4 Resumo

Este capítulo apresentou uma visão geral da abordagem, mostrando desde o sua fundamentação, seu fluxo até a arquitetura do monitor. A visão geral da abordagem mostrou o seu princípio básico de funcionamento, mostrou a sequência de passos que compõem o se fluxo de uso, desde a especificação de modelos TDevC até a execução da plataforma.

5 DEFINIÇÃO DA FSM-MONITOR

A abordagem proposta inclui um mecanismo de especificação de máquinas de estados para representar o protocolo de comunicação de alto nível de abstração e restrições impostas a este protocolo, as quais serão validadas durante a execução da plataforma.

Como já mencionado, uma FSM-Monitor é uma máquina de estados que representa o protocolo de comunicação entre processador e dispositivos. Cada estado desta máquina está associado a um *trace* na execução deste protocolo, onde este estado pode ser compostos também por asserções. Como será possível visualizar neste capítulo, a FSM-Monitor é bastante semelhante ao formalismo *Statecharts* (HAREL, 1987). Durante o desenvolvimento da FSM-Monitor, foi cogitado o uso de *Statecharts* ou alguma extensão de *Statecharts* para esta tese. Porém, pelo fato da FSM-Monitor ter uma aplicação a um domínio bem específico, opinou-se por construir o próprio formalismo. Isto permitiu moldar a máquina de estados exatamente para a abordagem pesquisada e reduziu o risco de que, por alguma restrição do formalismo *Statechart*, a abordagem não conseguisse alcançar os objetivos almejados.

As asserções associadas aos estados são especificadas através de fórmulas de lógica temporal (LTL) e são representadas através de autômatos Büchi (GASTIN; ODDOUX, 2001). Autômatos Büchi são máquinas de estados que estendem os autômatos finitos para suportarem entradas infinitas.

Quando as restrições são especificadas e transformadas em autômatos Büchi, isso significa que a execução do protocolo deve passar por pelo menos um estado final desta restrição infinita vezes. Como a execução de uma plataforma é algo finito, nega-se a especificação da restrição para que, quando a execução passar pelo menos uma única vez por um estado final, significa que a restrição foi violada. Em outras palavras, a restrição não foi atendida pelo menos uma vez.

Como uma FSM-Monitor pode ser hierárquica, cada estado pode conter uma ou mais máquinas de estados hierárquicas, consequentemente contendo um ou mais estados. Quando, por exemplo, dois estados A e B quaisquer pertencem a uma sub-máquina relacionada a um estado G, diz-se que os estados A e B são irmãos, uma vez que eles têm o mesmo pai, filhos de G e G é pai de A e B. A raiz de uma FSM-Monitor é sempre um único estado ancestral de todos os estados da máquina hierárquica, o qual é chamado de *Estado Global*.

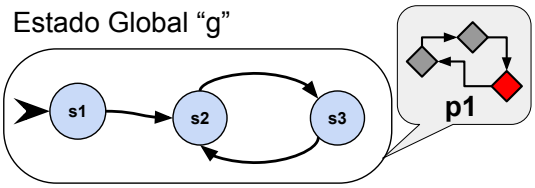


Figura 33 – Exemplo Hipotético de uma FSM-Monitor

A Figura 33 mostra um exemplo de uma FSM-Monitor proposta por este trabalho. Neste exemplo pode-se perceber a existência de 4 estados, chamados de *s1*, *s2* e *s3*, mais o Estado Global *g*. Além dos estados, esta máquina de estados tem associada ao estado *g* uma propriedade a ser observada durante a comunicação entre CPU e o respectivo dispositivo.

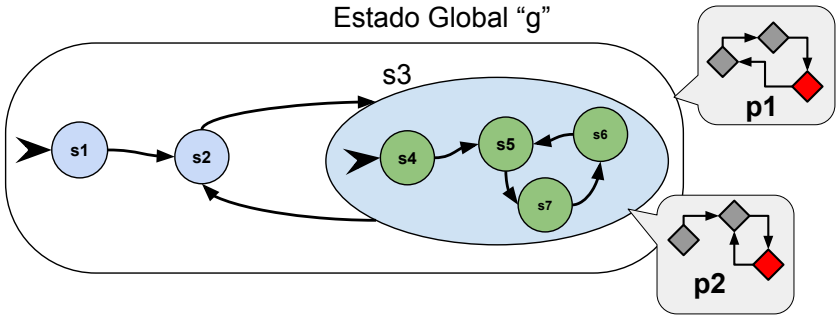


Figura 34 – Exemplo Hipotético de uma FSM-Monitor após um primeiro refinamento

Como proposto por esta abordagem, devido à hierarquia da FSM-Monitor é possível realizar o refinamento do protocolo e das propriedades de maneira incremental. Assim, supondo que a máquina de estados mostrada na Figura 33 seja refinada para a máquina de estados mostrada na Figura 34, o estado *s3* agora passa a ter 4 estados filhos (*s4*, *s5*, *s6* e *s7*) e um propriedade associada (*p2*). É importante destacar que, para o estado *s3* e seus filhos, tanto a propriedade *p1* quanto a propriedade *p2* têm efeito. Devido à hierarquia, as propriedades dos pais devem ser respeitadas dentro dos seus estados filhos.

Até esta etapa do refinamento a máquina de estados suporta apenas um contexto (sessão) da comunicação ou modo de operação, ou seja, apenas um único canal de comunicação com um fluxo de dados sequencial está sendo descrito. Para permitir mais sessões de comunicação é preciso especificar linhas de execução concorrentes de protocolo, representando contextos concorrentes de comunicação. A Figura 35 mostra como resolver este problema.

A FSM-Monitor da Figura 34 então é incrementada para a máquina de estados da Figura 35, onde máquinas de estados concorrentes podem sem observadas. Cada região que contém uma máquina de estado concorrente é chamada de Região Ortogonal. Ainda neste mesmo exemplo pode-se observar a existência de 4 Regiões Ortogonais, sendo duas 2 delas (*o1* e *o2*) pertencentes ao estado *g* e as outras duas (*o3* e *o4*) ao estado *s3*.

Dentro de cada estado pai, ou seja, aqueles que, na perspectiva de hierarquia, não são folhas na árvore da hierarquia, podem existir uma ou mais sub-máquinas de estados concorrentes e distintas que refletem linhas de execução simultâneas (*traces* concorrentes). Dentro de um estado, cada sub-máquina concorrente se encontra em uma *Região Ortogonal* distinta. Assim, cada estado não-folha pode conter uma ou várias Regiões Ortogonais contendo, cada uma delas, uma máquina de estados distinta e de execução concorrentes.

As Regiões Ortogonais foram explicitamente definidas na linguagem *TDevC* com o objetivo de impedir diretamente na descrição dos modelos que transições entre sub-

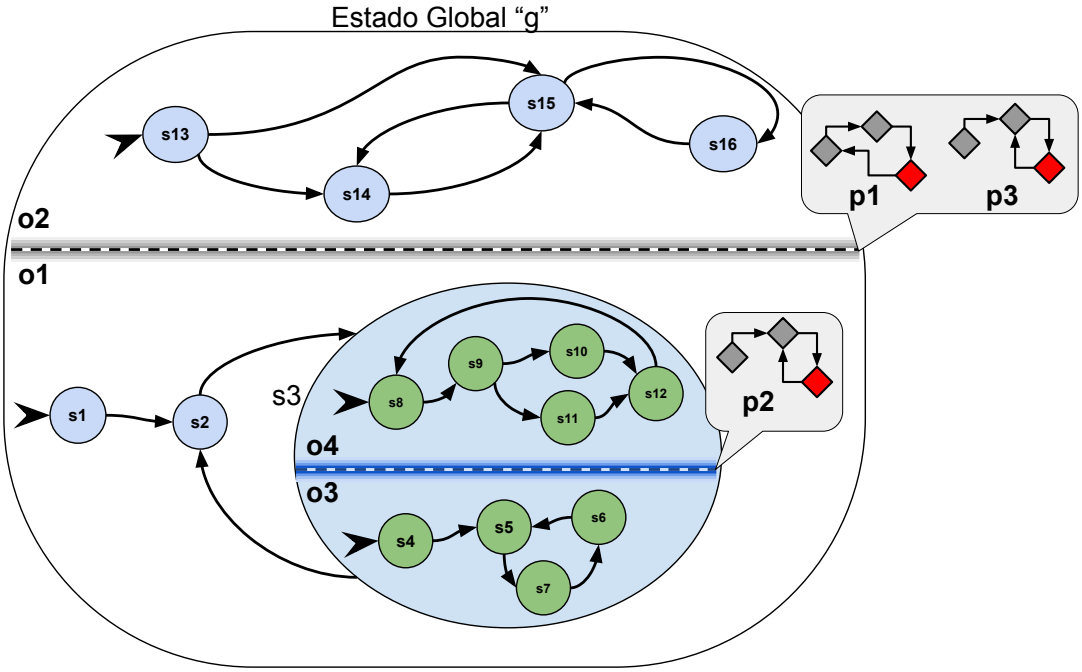


Figura 35 – Exemplo Hipotético de uma FSM-Monitor após um segundo refinamento

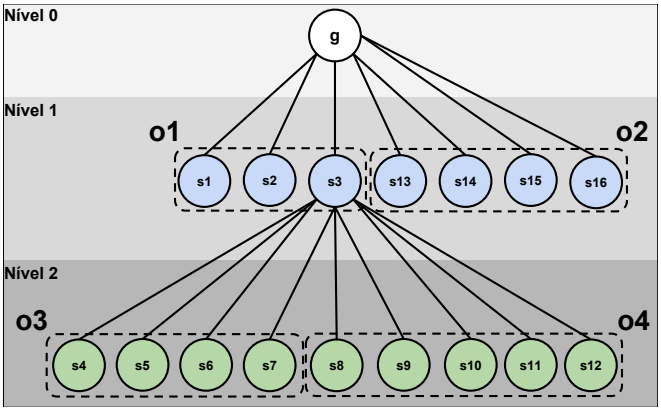


Figura 36 – Visão hierárquica do exemplo hipotético da FSM-Monitor

máquinas de diferentes estados sejam realizadas. Isto remove completamente a possibilidade de haver junções (*join*) ou separações (*fork*) nas linhas de execução concorrentes de diferentes sub-máquinas de estados e nem em estados em diferentes níveis de hierarquia.

Como cada máquina de estados existente em uma Região Ortogonal necessita de um estado inicial, mais uma classe de estados, chamada de *Estados Iniciais*, foi definida. No exemplo da Figura 35, os estados iniciais são os estados *s1*, *s4*, *s8* e *s13*.

Outro elemento que foi adicionado nesta nova especificação foi a propriedade *p3*. Agora, as propriedades *p1* e *p3* são diretamente associadas ao estado *g* e, consequentemente, estão associadas a todos os outros estados da FSM-Monitor. Já a propriedade *p2* está diretamente relacionada ao estado *s3* e indiretamente associada aos seus filhos: *s4*, *s5*, *s6* e *s7*, *s8*, *s9*, *s10*, *s11* e *s12*. Para todos os demais estados filhos direto do estado *g*, *p2* não tem efeito algum.

Para deixar ainda mais clara a relação de níveis de hierarquia mencionada, a Figura 36 mostra em uma perspectiva hierárquica o exemplo da Figura 35. Nesta figura é possível observar o Estado Global *g* no topo e consequentemente no nível 0. Logo abaixo, no nível 1, estão suas duas Regiões Ortogonais (*o1* e *o2*) e os seus estados filhos (*s1* até *s3* e *s13* até *s16*). Seguindo, no nível 2, as regiões ortogonais de *s3* (*o3* e *o4*) e os seus estados filhos e estados netos de *g* (*s4* até *s12*).

De uma maneira geral e simplificada para uma primeira explicação sobre os contratos definidos nesta abordagem, adota-se que tanto a FSM-Monitor quanto as máquinas Büchi representam *traces* de acessos realizados por um *device driver* ao seu respectivo dispositivo. Entretanto os *traces* especificados na FSM-Monitor servem puramente como um guia do protocolo para que o monitor tenha ciência em que estado da execução o dispositivo se encontra. Cada acesso equivale a um evento que pode disparar uma transição entre estados na FSM-Monitor, quando isso reflete uma mudança do estado do dispositivo também. Assim, cada estado desta máquina de estados contém, além das transições para outros estados, uma transição implícita para si conhecida como "*else*". Como o próprio nome diz, esta transição sempre ocorre quando nenhuma outra transição especificada é disparada. Assim, esta transição representa um evento indiferente para a mudança do estado do execução do dispositivo.

É importante chamar a atenção para o fato de que a especificação da FSM-Monitor é realizada de maneira incremental. Dessa maneira, quando o modelo estiver sendo descrito em um alto nível de abstração, nem todos os eventos irão disparar transições de estados. Em altos níveis de abstração os modelos podem ignorar algum eventos que são irrelevantes naquele momento. Por este motivo existem, de maneira transparente para o projetista, as transições "*else*".

Como a tradução do seu nome já deixa clara, esta transição é realizada toda vez que um evento não é representado por nenhuma das transições existentes do estado em questão. Isso quer dizer 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. O evento pode não representar uma mudança no estado de execução do dispositivo, como descrito na especificação do seu fabricante, mas pode ser um comando que o levará a um estado indesejado naquele ponto de execução. Justamente neste ponto que os *traces* representados pelos autômatos Büchi são monitorados. Estes *traces* representam puramente restrições da comunicação entre *drivers* e periféricos em pontos específicos do protocolo (estados da FSM-Monitor) as quais se desejam identificar durante a execução da plataforma.

Com esta forma de expressar *traces*, é possível então separar claramente protocolos básicos de comunicação para o uso de um periférico (suposições de um contrato) de propriedades comportamentais da execução desta comunicação associados à pontos específicos

destes protocolos (garantias de um contrato).

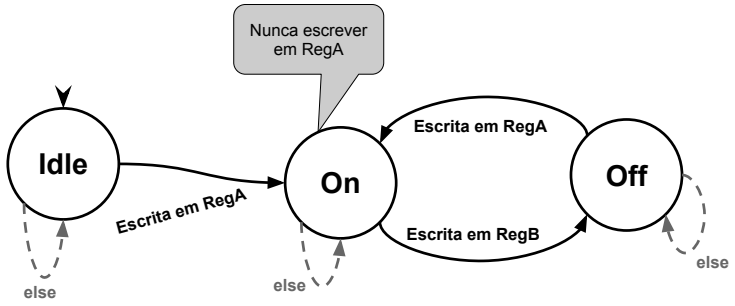


Figura 37 – Exemplo da diferenciação entre as suposições (estados e transições) e garantias (propriedade no estado "On") dos contratos aplicados aos protocolos de comunicação

A Figura 37 mostra um exemplo prático da separação entre as suposições e as garantias. Neste exemplo hipotético um dispositivo é inicializado em um modo inativo representado através do estado *Idle*. Assim que uma escrita ao seu registrador *RegA* é realizada, o dispositivo passa a operar usando o valor escrito em *RegA* com sendo a base de seu trabalho e supostamente indo para o estado ativo (*On*). Para desligá-lo basta apenas escrever qualquer valor no registrador *RegB* que supostamente o dispositivo irá para o estado *Off*.

Entretanto, existe uma restrição de uso; como o dispositivo utiliza o valor escrito em *RegA* para operar e pode usar este próprio registrador como base para algum cálculo, por exemplo, de acordo com sua especificação formal *"não é permitido em hipótese alguma escrever neste registrador enquanto o dispositivo estiver ativo, sob pena de não haver garantia da corretude dos resultados apresentados no fim da operação"*.

É importante destacar que os dispositivos recebem comando de maneira sequencial. Dessa maneira, não há concorrência na entrada de comandos. A única maneira do registrador *RegA* sofrer uma modificação externa é através de sua interface. Em outras palavras, pela natureza dos dispositivos anexados à plataforma, sua interface (além do próprio barramento) provê a exclusão mútua de acessos externos concorrentes.

Do ponto de vista do protocolo de comunicação, esta restrição não o afeta diretamente, uma vez que o *driver* tem todo direito e possibilidade de efetuar essa escrita em qualquer momento da execução do sistema e esta escrita não o levará a uma nova etapa do protocolo ou um novo estado de operação. Porém, para este caso especifico e neste ponto do protocolo, esta escrita não deve ser feita. Logo, supondo o ponto exato da execução do protocolo (estado) descrito na especificação do dispositivo, uma garantia (propriedade) foi imposta exatamente neste ponto.

A FSM-Monitor ainda suporta o uso de variáveis para auxiliar na definição dos protocolos e propriedades, quando necessário. O uso de variáveis dá suporte ao monitor realizar a validação com um contexto momento associado. Isto torna o monitoramento *stateful*.

A próxima seção tem como objetivo formalizar a semântica operacional da FSM-Monitor, o que servirá de fundamento para a apresentação da linguagem TDevC, tanto para a sua sintaxe quanto para a sua semântica.

5.1 Formalismo da FSM-Monitor

A FSM-Monitor, sendo uma máquina de estados, é essencialmente composta por um conjunto finito de estados, e um conjunto de funções que definem a relação entre esses estados. Esta seção irá formalizar este tipo de máquina de estados e, para auxiliar no entendimento deste formalismo, um dispositivo *Ethernet* será utilizado como exemplo. A medida que os conceitos foram sendo apresentados, o exemplo vai sendo mais detalhado.

O dispositivo Ethernet utilizado foi o DM9000A (DAVICOM, 2006; DAVICOM, 2005). O Ethernet DM9000A é um dispositivo que permite a comunicação entre equipamentos eletrônico através de cabo de par trançados. Este dispositivo especificamente contém dois bancos de registradores e um módulo, chamado *PHY*, responsável pela implementação em hardware da camada física de rede do modelo OSI. Esta módulo é o responsável pelo envio e recebimento físicos dos dados transmitidos pelo e para o dispositivo.

Assim, seja H uma FSM-Monitor, ela representada através de uma 12-tupla, tal que $H = \{Q \cup \{globalstate\} \cup \{q_r\}, R, L \cup \{q_r\}, F, P, B, C, J, T, E, \delta, M\}$, onde:

- Q representa todos os estados declarados da FSM-Monitor. Estes estados tem relação direta com protocolo especificado.
- $globalstate$ é o estado ancestral de todos os estados declarados.
- q_r é o estado de rejeição da FSM-Monitor. Este estado passa a ser o único estado corrente após a violação de alguma assertiva.
- R é o conjunto de todas as regiões ortogonais da FSM-Monitor H.
- L é o conjunto de todos os estados dos Autômatos Büchi gerados a partir das fórmulas LTL.
- F é o conjunto de todos os estados finais dos Autômatos Büchi gerados a partir das fórmulas LTL.
- P é o conjunto de todos os estados correntes da FSM-Monitor, tal que $P \subset Q \cup \{globalstate\} \cup \{q_r\}$.
- B é o conjunto de todos os estados correntes dos Autômatos Büchi gerados a partir das fórmulas LTL, tal que $B \subset L \cup \{q_r\}$.
- C é o conjunto de todos os estados finais dos Autômatos Büchi gerados a partir das fórmulas LTL que representam assertivas do tipo *CRITICAL*, tal que $C \subset F$.

- J é o conjunto das funções de definição de relação de elementos da FSM-Monitor, tal que $J = \{father, owner, rg, init, mem, sto\}$. Estas funções serão definidas e detalhadas mais adiante neste capítulo.
- T é o conjunto de todos os eventos que podem disparar transições de estados na FSM-Monitor.
- E é o conjunto de todos as expressões Booleanas que definem as condições para que haja uma transição de estados.
- δ é o sistema de transição que define as relações de transição entre os estados da FSM-Monitor.
- M o conjunto que representa uma memória da FSM-Monitor, tal que M = $\{(\mathbb{Z}, \mathbb{Z})\}$.

Para contextualizar o formalismo, assumamos que a FSM-Monitor H especifica o protocolo de comunicação entre um processador e o dispositivo Ethernet DM9000A. Assim, Seja Q o conjunto de todos os estados declarados, diz-se então que os elementos de Q representam os estados de execução do dispositivo Ethernet. O dispositivo *DM9000A* contém um módulo de controle da camada física de rede, o módulo *PHY*. Este módulo tem 2 estados básicos de execução: *PHYUP* e *PHYDOWN*. Logo, H é formada por um conjunto $Q = \{PHYUP, PHYDOWN\}$, onde *PHYUP* representa que a camada física está ativa e pronta para trocar dados com outro dispositivo externo, e *PHYDOWN* representa que o módulo *PHY* está inativo, interrompendo a troca de dados com o meio externo, como pode ser visto na Figura 38.

Figura 38 – Exemplo simplificado da FSM-Monitor H modelada para a Ethernet DM9000A

Porém, nem sempre os dispositivos contêm estados de operação simplificados e, em muitas vezes, esses estados de operação podem ser descritos como um conjunto de sub-operações.

Como dito anteriormente, no dispositivo DM9000A, quando a camada física é ativada, o dispositivo está pronto para transmitir e receber dados do meio externo. Assim, no estado *PHYUP*, além de apenas indicar que o *PHY* está ativo, indica também que o dispositivo está em um estado pronto para receber e transmitir dados. Pode-se dizer então que o estado *PHYUP* é formado por outros estados, ou estados filhos, como pode ser visto na Figura 39. Nota-se que, para o dispositivo DM9000A, quando este se encontra no estado *PHYUP*, ele

pode consequentemente se encontrar em um dos estados *DataWrite*, *SetLen*, *SetReady*, *WaitReady*, *SendData*, *WaitData* e *ReadData*. Assim, o conjunto de estados de H passa a ser $Q = \{PHYUP, PHYDOWN, DataWrite, SetLen, SetReady, WaitReady, SendData, WaitData, ReadData\}$.

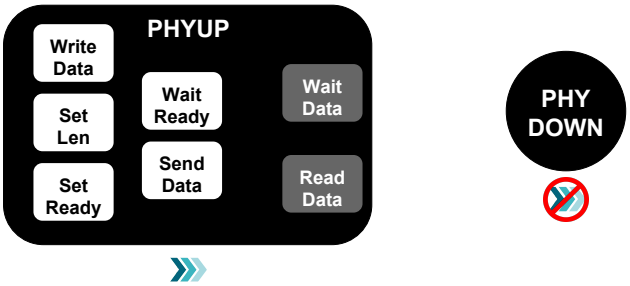


Figura 39 – Exemplo simplificado da FSM-Monitor H com hierarquia de estados modelado para a Ethernet DM9000A

5.1.1 Relação entre Estados

Apesar da existência de diversos estados na especificação dos dispositivos, ainda é preciso definir uma relação de parentesco entre eles e, particularmente para H, entre o estado $PHYUP$ e os estados *DataWrite*, *SetLen*, *SetReady*, *WaitReady*, *SendData*, *WaitData* e *ReadData*. Definiu-se então uma função $father : Q \to Q$ que mapeia um estado filho a seu estado pai, tal que a função $father \in J$. Assim, para H, tem-se que:

- $father(DataWrite) = PHYUP$
- $father(SetLen) = PHYUP$
- $father(SetReady) = PHYUP$
- $father(WaitReady) = PHYUP$
- $father(SendData) = PHYUP$
- $father(WaitData) = PHYUP$
- $father(ReadData) = PHYUP$

E para a função inversa $father^{-1} : Q \to Q$ tem-se:

- $father^{-1}(PHYUP) = \{DataWrite, SetLen, SetReady, WaitReady, SendData, WaitData, ReadData\}$

Por questões de organização e agrupamento de características gerais dos estados da máquina, toda *FSM-Monitor* é composta por pelo menos um estado, conhecido como

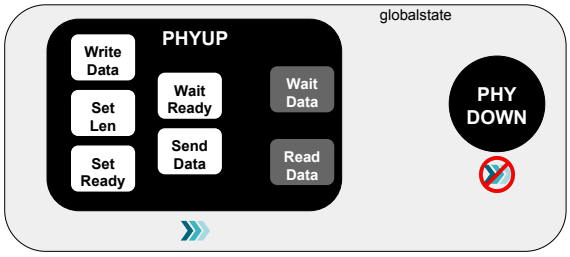


Figura 40 – Exemplo simplificado da FSM-Monitor com estado global modelado para a Ethernet DM9000A

globalstate, o qual é ancestral de todos os estados da máquina, com exceção dele mesmo, como pode ser visto na Figura 40.

Logo a função *father* precisa ser modificada para suportar essa característica. Assim, tem-se que $father : Q - \{globalstate\} \rightarrow Q$, uma vez que o *globalstate* não tem pai. Com o estado *globalstate*, nenhum outro estado pode ser ancestral de si próprio. Em outras palavras, não há ciclos na hierarquia dos estados, de maneira que sempre $father_1(father_2(...father_n(q)...)) \neq q \mid q \in Q$. Dessa maneira pode-se definir a primeira propriedade da FSM-Monitor:

$$father_n(q) \neq q, \forall n \geq 1 \wedge \forall q \in Q. \tag{5.1}$$

Para o dispositivo Ethernet, quando a camada física está ativa, o conjunto dos estados *DataWrite, SetLen, SetReady, WaitReady e SendData* dita sua execução para o envio e o conjunto dos estados *WaitData e ReadData* para o recebimento de dados. Entretanto, existe uma sequência de lógica entre esses estados, especificada pelo fabricante do dispositivo, que define corretamente como a ethernet deve ser controlada para que a transmissão e o recebimento de dados sejam realizados de maneira correta.

O dispositivo DM9000A, para enviar uma carga útil de dados via cabo de par trançado, requer que (i) estes dados sejam escritos na memória interna do dispositivo pelo processador. Em seguida, o elemento de processamento deve (ii) registrar o tamanho da carga útil escrita na memória interna da DM9000A para, por fim, o elemento de processamento (iii) sinalizar que os dados já estão configurados e prontos para serem enviados.

Assim que o elemento de processamento sinaliza que os dados estão prontos, o dispositivo DM9000A (iv) verifica que os dados estão prontos e então (v) inicia o envio dos dados pelo cabo de par trançado.

As etapas (i), (ii), (iii), (iv), (v) do envio de dados pelo dispositivo DM9000A descritas n parágrafo anterior são relacionados diretamente com os estados *DataWrite*, *SetLen*, *SetReady*, *WaitReady*, *SendData* da Figura 39 e Figura 40.

Dessa maneira, além de relações hierárquicas entre estados, existe também a relação de transição descrita através de um sistema de transição apoiado em uma série de funções e propriedades que serão detalhadas mais adiante neste documento. A priori é importante

apenas entender que existem transições entre estados que são disparadas diante da ocorrência de eventos externos ao dispositivo (clock, sinais de externos, comandos de usuários, etc.). A Figura 41 mostra a FSM-Monitor para o dispositivo DM9000A com as relações de transição entre seus estados.

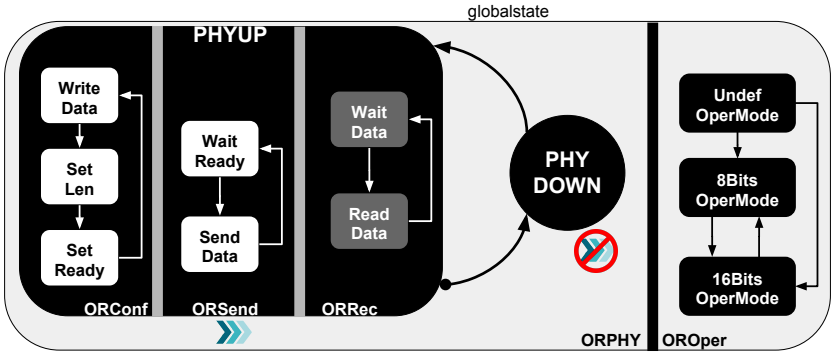


Figura 41 – Exemplo FSM-Monitor com regiões ortogonais modelado para a Ethernet DM9000A

Na Figura 41 é possível perceber que as transições dentro do estado *PHYUP* podem ocorrer concorrentemente em três linhas de execução. A primeira descrevendo a escrita e configuração dos dados na memória do dispositivo, a segunda descrevendo o envio dos dados via cabo de par trançado e a terceira descrevendo o recebimento dos dados externos, via cabo de par trançado. Isto é bastante comum em dispositivos, quando várias sub-operações independentes ocorrem concorrentemente alternando-as entre os eventos.

A escrita e configuração dos dados foi descrita de maneira independente e concorrente com o envio dos dados porque, de acordo com o manual de aplicação do dispositivo (DAVICOM, 2005), o dispositivo DM9000A suporta que duas cargas uteis de dados distintas sejam trabalhadas concorrentemente. Enquanto um volume de dados está sendo transmitido, outro volume de dados distinto pode ser escrito na memória do dispositivo, simultaneamente.

A Figura 41 também mostra outra linha de execução distinta da ativação ou desativação da camada física. Esta outra linha, contendo os estados *UndefOperMode*, *8BitsOperMode* e *16BitsOperMode*, descreve o modo de operação do dispositivo. Novamente, de acordo com o manual de aplicação do dispositivo (DAVICOM, 2005), o dispositivo DM9000A suporta 2 modos de operação distintos, onde o modo de operação de 8 bits diz que o dispositivo está conectado à uma plataforma através um meio de comunicação físico cuja palavra tem um tamanho de 8 bits e o modo de operação de 16 bits, um meio de comunicação físico cuja palavra tem um tamanho de 16 bits. O estado *UndefOperMode* representa o momento antes do *device driver* requisitar a informação sobre em que modo de operação a Ethernet está operando. Em outras palavras, o momento em que, para o *driver*, o modo de operação ainda é indefinido.

Assim, até o momento, o conjunto de estados passa a ser Q={*PHYUP*, *PHYDOWN*, *DataWrite*, *SetLen*, *SetReady*, *WaitReady*, *SendData*, *WaitData*, *ReadData*, *UndefOperMode*,

8BitsOperMode, *16BitsOperMode*}.

Quando há a especialização explícita das linhas concorrentes de execução de sub-operações evita-se um número excessivo de transições e a explosão do número de estados, além de deixar a formalização do modelo muito mais clara e próxima do comportamento real do dispositivo.

Por este motivo, na FSM-Monitor, os estados pai podem ser segmentados em um ou mais regiões contendo estados e transições entre estes estados, os quais representam linhas de execuções independentes e concorrentes. Essas regiões são conhecidas como Regiões Ortogonais, e R é o conjunto de todas as regiões ortogonais de uma FSM-Monitor. A existência de pelo menos uma Regiões Ortogonais é obrigatória.

Logo, para que a relação de hierarquia de estados tenha uma formalização mais detalhada, envolvendo todos os elementos da FSM-Monitor, se faz necessário relacionar as estados pais com suas regiões ortogonais e também os estados filhos com as regiões ortogonais às quais eles estão contidos. A função $owner \in J$, tal que $owner : R \rightarrow Q$, mapeia uma região ao estado no qual ela está contida. Assim, para H tem-se que:

- $owner(ORPHY) = globalstate$
- $owner(OROper) = globalstate$
- $owner(ORConf) = PHYUP$
- $owner(ORSend) = PHYUP$
- $owner(ORRec) = PHYUP$

E para a função inversa $owner^{-1} : Q \rightarrow R$ tem-se:

- $owner^{-1}(globalstate) = \{ORPHY, OROper\}$
- $owner^{-1}(PHYUP) = \{ORConf, ORSend, ORRec\}$

Já a função $rg \in J$, onde $rg : Q - \{globalstate\} \rightarrow R$, mapeia os estados às regiões ortogonais nas quais eles estão contidos. O leitor deve perceber que o *globalstate* não faz parte do domínio da função *rg*. Isso se dá pelo fato de que o *globalstate* não está contido em nenhuma região ortogonal, uma vez que ele é o ancestral de todo os outros estados do uma FSM-Monitor. Para H, *rg* é:

- $rg(PHYUP) = rg(PHYDOWN) = ORPHY$
- $rg(DataWrite) = rg(SetLen) = rg(SetReady) = ORConf$
- $rg(WaitReady) = rg(SendData) = ORSend$
- $rg(WaitData) = rg(ReadData) = ORRec$

- $rg(UndefOperMode) = rg(8BitsOperMode) = rg(16BitsOperMode) = OROper$

E para a função inversa $rg^{-1} : R \rightarrow Q - globalstate$ tem-se:

- $rg^{-1}(ORPHY) = \{PHYUP, PHYDOWN\}$
- $rg^{-1}(ORConf) = \{DataWrite, SetLen, SetReady\}$
- $rg^{-1}(ORSend) = \{WaitReady, SendData\}$
- $rg^{-1}(ORRec) = \{WaitData, ReadData\}$
- $rg^{-1}(OROper) = \{UndefOperMode, 8BitsOperMode, 16BitsOperMode\}$

Utilizando estas duas funções, pode-se então definir mais uma propriedade da FSM-Monitor que refina a relação de hierarquia, mas desta vez através das regiões ortogonais:

$$father(q) = owner(rg(q)) \mid \forall q \in Q - \{globalstate\} \tag{5.2}$$

A equação (5.2) mostra que o estado pai de um estado q é mesmo estado dono da região ortogonal na qual o estado q está incluído.

Como todo dispositivo digital tem um estado inicial de operação conhecido, os estados de uma FSM-Monitor podem ser considerados iniciais ou intermediários. Os estados iniciais são uma referência para indicar qual estado de operação o dispositivo deve se encontrar em situações iniciais, sejam elas, por exemplo, quando este é inicializado ou quando uma operação composta de sub-operações é iniciada após uma transição de estado. Os demais são considerados intermediários. No exemplo da DM9000A, definiu-se que o dispositivo sempre é iniciado nos estados *PHYDOWN* e *UndefOperMode*. Quando a camada física é ligada através de um comando partindo do *device driver*, o estado de operação *PHYUP* é inicializado nos estados *DataWrite*, *WaitReady* e *WaitData*, e, quando o *driver* requisita a informação sobre o modo de operação, há uma transição para os estados *8BitsOperMode* ou *16BitsOperMode*. Estes estados iniciais podem ser vistos na Figura 42, sinalizados através de uma seta semelhante a uma ponta de flecha.

Assim, é preciso definir uma função que mapeia os estados iniciais às suas respectivas regiões ortogonais. É importante deixar claro que um estado pode conter mais de um estado filho inicial, entretanto cada um corresponde a apenas uma única região ortogonal. Por isso, é necessário mapear essa relação de região ortogonal com seus estados iniciais. Foi então definida a função $init \in J$, tal que $init : R \rightarrow Q - globalstate$. Assim, para H tem-se que:

- $init(ORPHY) = PHYDOWN$
- $init(OROper) = UndefOperMode$

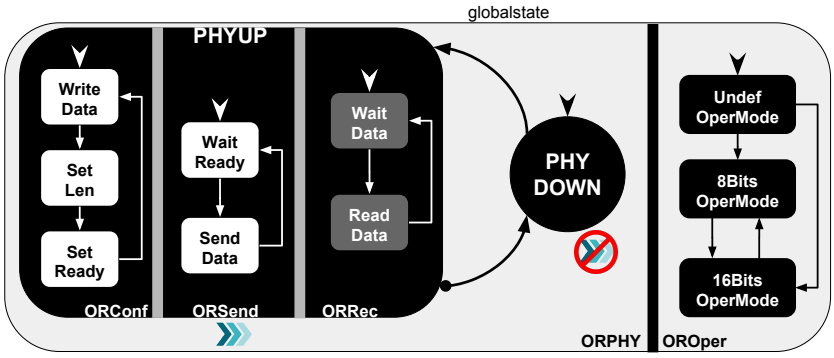


Figura 42 – Exemplo da FSM-Monitor com estados iniciais modelado para a Ethernet DM9000A

- $init(ORConf) = WriteData$
- $init(ORSend) = WaitReady$
- $init(ORRec) = WaitData$

E para a função inversa $init^{-1} : Q - globalstate \rightarrow R$ tem-se:

- $init^{-1}(PHYDOWN) = ORPHY$
- $init^{-1}(UndefOperMode) = OROper$
- $init^{-1}(WriteData) = ORConf$
- $init^{-1}(WaitReady) = ORSend$
- $init^{-1}(WaitData) = ORRec$
- $init^{-1}(PHYUP) = init^{-1}(8BitsOperMode) = init^{-1}(16BitsOperMode) = init^{-1}(SetLen) = init^{-1}(SetReady) = init^{-1}(SendData) = init^{-1}(ReadData) = \emptyset$

5.1.2 Transição de Estados

Como mencionado, há uma relação de transição entre os estados. Esta relação dá-se início nos estados iniciais e não há estados finais, uma vez que a execução de um dispositivo é contínua. Estas transições são disparadas por um evento externo: acesso à memória do dispositivo (registradores). Entretanto, nem sempre um acesso a um registrador dispara a ocorrência de transições. Cada transição de estado tem sua própria condição de ocorrência. Assim, toda vez que o evento é disparado, todas as transições do estado corrente têm suas condições checadas e, caso alguma seja verdadeira, a transição ocorre. A Figura 43 mostra a FSM-Monitor H com as transições entre os estados $PHYDOWN$ e $PHYUP$ e suas expressões $WRITE(GPR[0]) = 0$ e $WRITE(GPR[0]) = 1$. Estas expressões, respectivamente, representam uma escrita do valor 0 no bit 0 do registrador GPR e uma escrita do valor 0 no bit 1 do registrador GPR.

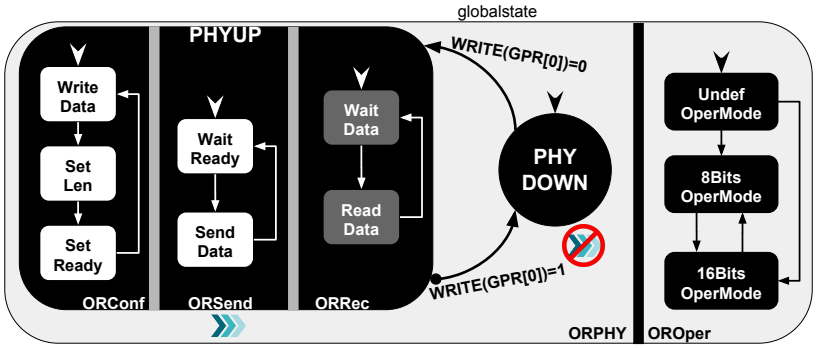


Figura 43 – Exemplo da FSM-Monitor com transições de estados modelado para a Ethernet DM9000A

Pode-se então definir um sistema de transição simplificado, onde a partir de um estado corrente $p \in Q$, após um evento $\tau \in T$, sendo T o conjunto de todos os eventos possíveis, e uma expressão Booleana válida e, onde $e \in E$, é possível alcançar um outro estado $q \in Q$, onde após o evento, que passa a ser o estado corrente. Assim este sistema de transição, chamado de *deltas*, é definido a seguir:

$$(p) \overset{e}{\underset{\tau}{\Rightarrow}} (q) \tag{5.3}$$

Com base neste sistema de transição simplificado, tem-se as seguintes transições para a FSM-Monitor H:

- $(PHYDOWN) \xrightarrow[\tau_i]{WRITE(GPR[0])=0} (PHYUP)$
- $(PHYUP) \xrightarrow[\tau_j]{WRITE(GPR[0])=1} (PHYDOWN)$

Antes de entrar em detalhe sobre as expressões das transições, é importante descrever com mais detalhes os eventos de transições. Um elemento $\tau \in T$ é representado através de uma 3-tupla onde $\tau = \{t, a, d\}$, onde t representa o tipo acesso (escrita ou leitura em registrador), a representa o endereço relativo do registrador e d, o valor do dado lido ou escrito.

Assim, o conjunto finito $T = \{\tau_1, \tau_2, ..., \tau_k\}$ é composto por k elementos, onde k é o número de possibilidades de combinações de t, a e d, lembrando que, para um dispositivo real, a e d tem valores finitos de acordo com a especificação da arquitetura.

Como pode ser visto nas transições acima, para cada uma é especificado um evento τ distinto (τ_i e τ_i). Elas foram descritas dessa maneira para explicitar que estas duas transições são realizadas na ocorrência eventos diferentes.

É possível perceber pela Figura 43 que a transição do estado *PHYDOWN* para o *PHYUP* irá consequentemente resultar em uma transição automática para os estados iniciais filhos, uma vez que, assim que o estado *PHYUP* é referenciado como corrente, os seus estados filhos iniciais automaticamente também são.

Dessa maneira pode-se dizer que existe uma transição vazia entre o estado $PHYUP$ e seus estados filhos iniciais, sem a existência de um evento e expressão. A Figura 44 mostra a FSM-Monitor H todas as transições e expressão, incluíndo as transições vazias representadas através das setas pontilhadas. Perceba as três transições vazias (ε) no estado $PHYUP$ até os seus três estados iniciais. Dessa maneira pode-se estender o sistema de transição descrito em 5.3 para:

$$(p) \underset{\varepsilon}{\Rightarrow} (q) \tag{5.4}$$

Com base neste sistema de transição simplificado e representando as transições entre os estados $PHYUP$ e $PHYDOWN$, tem-se para H:

- $(PHYDOWN) \xrightarrow[\tau_1]{WRITE(GPR[0])=0} (PHYUP)$
- $(PHYUP) \xrightarrow[\tau_2]{WRITE(GPR[0])=1} (PHYDOWN)$
- $(PHYUP) \underset{\varepsilon}{\Rightarrow} (WriteData)$
- $(PHYUP) \underset{\varepsilon}{\Rightarrow} (WaitReady)$
- $(PHYUP) \underset{\varepsilon}{\Rightarrow} (WaitData)$

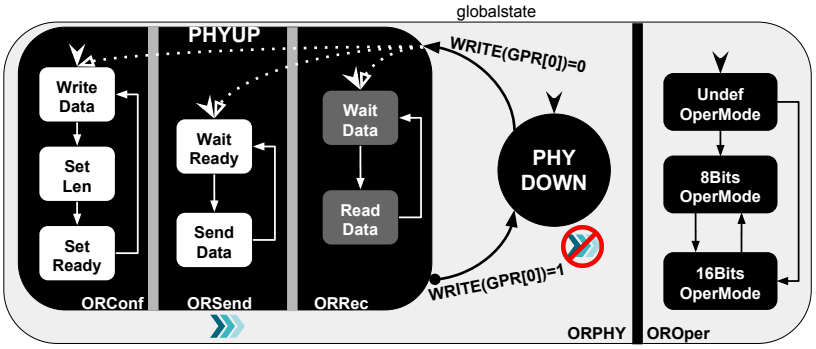


Figura 44 – Exemplo da FSM-Monitor com transições vazias modelado para a Ethernet DM9000A

A partir deste sistema de transição de H é possível perceber que, para alguns conjuntos de estados há uma transição com base no mesmo evento e respeitando a mesma expressão.

Assim, estendendo o sistema de transição descrito em 5.4, os estados correntes prévios e posteriores do sistema foram descritos como conjuntos justamente pelo fato de que existem transições que, a partir de um estado, alcança-se diversos estados, e também a partir de vários estados alcança-se um único estado. Além disso, podem existir diversos estados correntes simultaneamente, uma vez que a FSM-Monitor é formada por regiões ortogonais

hierarquicamente organizadas. Assim, o sistema de transição refinado pode ser definido como:

$$(P) \overset{\varepsilon}{\underset{\tau}{\Rightarrow}} (P') \tag{5.5}$$

sendo $P \subset Q$ e $P' \subset Q$. Agora o sistema de transição para H e representando apenas as transições entre os estados $PHYUP$ e $PHYDOWN$ tem-se:

- $(PHYDOWN) \xrightarrow[\tau_1]{WRITE(GPR[0])=0} (PHYUP)$
- $(PHYUP) \underset{\varepsilon}{\Rightarrow} (\{WriteData, WaitReady, WaitData\})$
- $(PHYUP, DataWrite, SetLen, SetReady, WaitReady, SendData, WaitData, ReadData\}) \xrightarrow[\tau_2]{WRITE(GPR[0])=1} (PHYDOWN)$

É importante destacar que, diferentemente de uma transição vazia, as transições de saída de um estado pai (por exemplo, de $PHYUP$ para $PHYDOWN$), compartilham os eventos e as expressões das transições do seu estado pai. Em outras palavras, é como se cada estado filho herdasse as transições do seu estado pai, sendo estas transições tendo como alvo os mesmo estados alvo que as transições do estado pai tem. Isso se dá pelo fato de que, se as transições de saída fossem transições vazias para seu estado pai, a qualquer momento os estados filhos correntes poderiam efetuar transições sem q houvesse evento algum.

As transições da FSM-Monitor, além permitirem a passagem de um estado para outro, elas permitem também manter informações sobre um contexto geral do dispositivo. Isso é possível pois a FSM-Monitor contém uma memória global, sendo então possível alocar variáveis e atribuir valores a elas em qualquer transição de estados. A Figura 45 mostra o exemplo de duas variáveis, *vUP* e *vDOWN*, que simplesmente mantém uma contagem do número vezes em que o dispositivo foi ligado e desligado. Perceba que essa atribuição é disparada toda vez que há uma transição entre os estados $PHYUP$ e $PHYDOWN$.

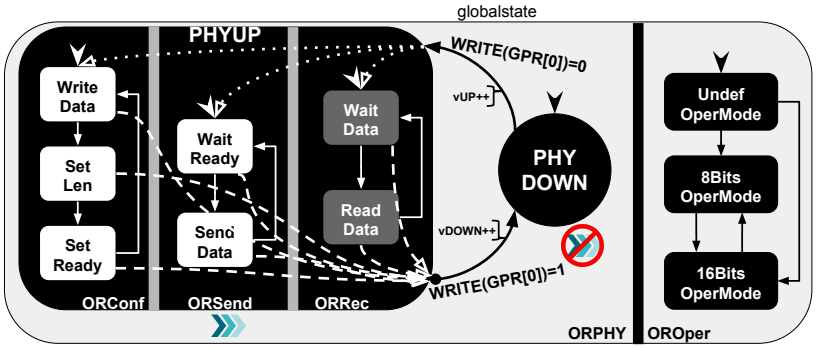


Figura 45 – Exemplo da FSM-Monitor com atribuições de variáveis modelado para a Ethernet DM9000A

Para manipular esta memória foram definidas então duas funções: $mem : \{M\} \times \mathbb{Z} \to \mathbb{Z}$ e $sto : \{M\} \times \mathbb{Z} \times \mathbb{Z} \to \{M\}$. A função *mem* mapeia um endereço inteiro da memória a um respectivo valor inteiro. Já a função *sto* vincula um valor inteiro a uma posição na memória mapeada através de um endereço inteiro. Esta função acaba retornando memória atualizada. Pode-se dizer que a função *sto* persiste dados na memória e a função *mem* os recupera. Assim, a função *sto* pode ser executada inúmeras vezes em uma transição, atualizando assim a memória.

Entretanto, para que a memória não possua mais de um valor associado ao mesmo endereço, a função *sto* faz uso da função *mem* durante a vinculação dos dados para consultar a memória e poder removendo o par *endereço/valor* desatualizado e e adicionar o novo. Assim, a função *sto* pode ser definida como:

$$sto(M, i, j) = sto(M' \dot{\cup} mem(M, i), j) = M' \cup (i, j) \mid i \in \mathbb{Z} \text{ e } j \in \mathbb{Z} \tag{5.6}$$

Dessa maneira, pode-se então estender o sistema de transição 5.5 para uma definição que contemple a atribuição de variáveis:

$$(P, M) \overset{e}{\underset{\tau}{\Rightarrow}} (P', sto_k(...sto_1(sto_0(M, r_0, v_0), r_1, v_1)..., r_k, v_k)) \tag{5.7}$$

Sendo o retorno de uma função *sto* um novo conjunto M representando a memória atualizada da FSM-Monitor, pode-se dizer que

$$\begin{aligned} sto_k(...sto_1(sto_0(M, r_0, v_0), r_1, v_1)..., r_k, v_k) &\iff \\ sto_k(...sto_1(M_0, r_1, v_1)..., r_k, v_k) &\iff \\ sto_k(...M_1..., r_k, v_k) &= M' \end{aligned} \tag{5.8}$$

É importante destacar que a atribuição de valores a variáveis é realizada de maneira sequencial e, dessa maneira, a função sto_0 representa a primeira atribuição de variável e sto_k representa a última atribuição. Desse modo, define-se o sistema de transição como sendo:

$$(P, M) \overset{e}{\underset{\tau}{\Rightarrow}} (P', M') \tag{5.9}$$

Agora, a partir de um conjunto de estados P e uma função *mem*, na ocorrência de um evento τ e sendo uma expressão e válida, será alcançado um novo conjunto de estados P' e uma função *mem* atualizada mem'.

Com esse sistema de transição é possível especificar algumas propriedades as quais todo o sistema deve respeitar. A primeira propriedade diz respeito à transição entre estados

folha, como pode ser vista na equação 5.10. Os estados folhas são estados que não são pais.

$$(P \dot{\cup} \{p\}, mem) \overset{e}{\underset{\tau}{\Rightarrow}} (P \cup \{q\}, M') \iff \begin{cases} (i)\ father^{-1}(p) = father^{-1}(q) = \emptyset \\ (ii)\ rg(p) = rg(q) \\ (iii)\ q \in P \Leftrightarrow p = q \end{cases} \tag{5.10}$$

Assim, para que essa transição seja válida é preciso que os estados p e q sejam folhas e que pertençam à mesma região ortogonal. Essa propriedade garante que não haja junção nem bifurcação de linhas de execução de diferentes regiões ortogonais.

Outro conjunto de propriedades diz respeito às transições entre estados folha e um estado pai. A equação 5.11 mostra este tipo de transição através do sistema de transição e define algumas propriedades que regem esta transição

$$\begin{aligned} (P \dot{\cup} \{p\}, M) \overset{e}{\underset{\tau}{\Rightarrow}} (P \cup \{q\}, M') \underset{\varepsilon}{\Rightarrow} \\ (P \cup \{q\} \cup \{init(r_0)\} \cup \{init(r_1)\} \cup ... \cup \{init(r_k)\}, M') \iff \end{aligned} \begin{cases} (i)\ father^{-1}(p) = \emptyset \\ (ii)\ father^{-1}(q) = rg^{-1}(owner^{-1}(q)) \neq \emptyset \\ (iii)\ rg(p) = rg(q) \\ (iv)\ q \notin (P \cup \{p\}) \\ (v)\ owner^{-1}(q) = \{r_0, r_1, ..., r_k\} \end{cases} \tag{5.11}$$

É possível perceber na equação 5.11 que existem duas transições em sequência, sendo a segunda uma transição vazia. Isso se dá pelo fato de que o estado alvo q deve ser um estado pai 5.11-(ii) e, dessa maneira, haverá uma transição vazia para os seus estados filhos iniciais. Mais uma vez, por questões de garantia de que não haverá junção e bifurcação, as propriedades 5.11-(iii) e (iv) são definidas.

O conjunto de propriedades descrito na equação 5.12 está relacionado a transições de saída, ou seja, transições entre um estado pai e um estado folha.

$$(P \dot{\cup} \{p_1, ..., p_n\} \dot{\cup} \{p\}, M) \overset{e}{\underset{\tau}{\Rightarrow}} (P \cup \{q\}, M') \iff \begin{cases} (i) \exists (P \dot{\cup} \{p\}, M) \overset{e}{\underset{\tau}{\Rightarrow}} (P \cup \{q\}, M') \\ (ii)\ \forall p_k \in \{p_1, ..., p_n\}, father(p_k) = p \\ (iii)\ father^{-1}(q) = \emptyset \\ (iv)\ rg(p) = rg(q) \\ (v)\ q \notin (P \cup \{p_1, ..., p_n\} \cup p) \end{cases} \tag{5.12}$$

Como um estado pai e seus filhos são simultaneamente estados correntes 5.12-(ii) e, se existe uma transição do estado pai para outro estado 5.12-(i), deve haver uma transição automática dos seus estados filhos para o mesmo estado alvo. A propriedade 5.12-(iv) garante mais uma vez que não haja join and fork entre as linhas de execução e a propriedade 5.12-(v) garante que não haja transições de um estado pai diretamente para um estado filho seu.

O conjunto de propriedades descrito na equação 5.13 é uma composição da equação 5.11 e 5.12. Estas propriedades descrevem uma transição entre dois estados pai (5.13-(ii) e (iii).

$$
\begin{aligned}
&(P \dot{\cup} \{p_1, ..., p_n\} \dot{\cup} \{p\}, M) \overset{e}{\underset{\tau}{\Rightarrow}} (P \cup \{q\}, M') \underset{\varepsilon}{\Rightarrow} \\
&\quad (P \cup \{q\} \cup init(owner^{-1}(q), M')) \iff \\
&\begin{cases}
(i) \exists (P \dot{\cup} \{p\}, M) \overset{e}{\underset{\tau}{\Rightarrow}} (P \cup \{q\}, M') \\
(ii)\ father(p_1) = ... = father(p_n) = p \\
(iii)\ \{p_1, ..., p_n\} \subset father^{-1}(p) \\
(iv)\ father^{-1}(q) \neq \emptyset \\
(v)\ rg(p) = rg(q) \\
(vi)\ q \notin (P \cup \{p_1, ..., p_n\} \cup p)
\end{cases}
\end{aligned}
\tag{5.13}
$$

Logo, é possível perceber que há uma transição de saída seguida por uma transição vazia. Dessa maneira, para que haja essa transição, deve existir uma transição entre os estados pai 5.13-(i). Como a equação 5.12, as propriedades 5.12-(iv) e (v) garantem que não haja junções e bifurcações entre as linhas de execução de diferentes regiões ortogonais e que não haja transição de um estado pai diretamente para um estado filho seu.

As equações seguintes (5.14 e 5.15) descrevem propriedades de transições que envolvem estados ancestrais. Essas propriedades definem como se comportam as transições dos estados descendentes, tanto em uma transição de saída (5.14) quanto em uma transição para um estado ancestral (5.15).

$$
\begin{aligned}
&(P \dot{\cup} \{p_i\} \dot{\cup} ... \dot{\cup} \{p_1\} \dot{\cup} \{p_0\}, M) \overset{e}{\underset{\tau}{\Rightarrow}} \\
&\quad (P \cup \{q\}, M') \iff \\
&\begin{cases}
(i) \exists (P \dot{\cup} \{p_0\}, M) \overset{e}{\underset{\tau}{\Rightarrow}} (P \cup \{q\}, M') \\
(ii)\ father(p_i) = p_{i-1} \forall i \geq 1 \\
(iii)\ father^{-1}(q) = \emptyset \\
(iv)\ rg(p_0) = rg(q) \\
(v)\ q \notin (P \cup \{p_i\} \cup ... \cup \{p_1\} \cup \{p_0\})
\end{cases}
\end{aligned}
\tag{5.14}
$$

Na equação 5.14, p_i é o estado folha descendente de p_0 no maior nível de hierarquia (5.14-(ii)). Sendo p_0 um estado na mesma região ortogonal de do estado q (5.14-(iv)) e q sendo um estado folha (5.14-(iii)), havendo uma transição entre p_0 e q (5.14-(i)), haverá também transições entre todos os ancestrais de p_0 e p. As propriedades (5.14-(iv) e (v)), mais uma vez, garantem a não ocorrência do surgimento de join and fork e a não ocorrência de transições de um estado pai diretamente para um estado filho seu.

A equação 5.15 cobre o caso de transições entre dois estados ancestrais. Havendo uma transição de saída do estado ancestral para um outro estado ancestral alvo ((5.15-(i)), como na equação 5.14, há transições de todos os estados descendentes automaticamente para o estado alvo. Na sequência, há transições vazias para todos os estados iniciais descendentes do estado alvo.

$$
\begin{aligned}
&(P \dot{\cup} \{p_i\} \dot{\cup} ... \dot{\cup} \{p_1\} \dot{\cup} \{p_0\}, M) \overset{e}{\underset{\tau}{\Rightarrow}} \\
&(P \cup \{q_0\}, M') \underset{\varepsilon}{\Rightarrow} \\
(P \cup \{q\} \cup init(owner^1(...init(owner^{-1}(q)...)), M') \iff & \\
&\begin{cases}
(i) \exists (P \dot{\cup} \{p_0\}, M) \overset{e}{\underset{\tau}{\Rightarrow}} (P \cup \{q\}, M') \\
(ii) father(p_i) = p_{i-1}, \ \forall i \geq 1 \\
(iii) \ father^{-1}(q) \neq \emptyset \\
(iv) \ rg(p_0) = rg(q) \\
(v) \ q \notin (P \cup \{p_i, ..., p_0\})
\end{cases}
\end{aligned}
\tag{5.15}
$$

5.1.3 Assertivas

Também na equação 5.15, p_i é o estado folha descendente de p_0 no maior nível de hierarquia (5.14-(ii). Sendo p_0 um estado na mesma região ortogonal de do estado q (5.15-(iv)) e q sendo um estado pai (5.15-(iii)), havendo uma transição entre p_0 e q (5.14-(i)), haverá também transições entre todos os ancestrais de p_0 e p, e transições vazias para todos os estados iniciais descendentes de q. As propriedades (5.15-(iv) e (v)), mais uma vez, garantem a não ocorrência do surgimento de junção e bifurcação entre diferentes regiões ortogonais e a não ocorrência de transições de um estado pai diretamente para um estado filho seu.

Com base nos eventos, estados e variáveis, algumas regras de contrato de uso do protocolo podem ser definidas. Essas regras de contrato são conhecidas neste trabalho como propriedades do protocolo de comunicação.

Como pode ser visto na Figura 46, duas propriedades de protocolo foram especificadas, sendo a propriedade $P1$ pertencente associada ao estado *globalstate* e a $P2$, ao estado *PHYUP*.

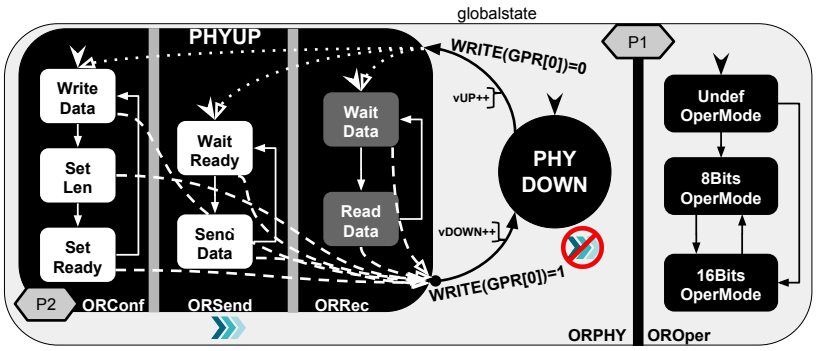


Figura 46 – Exemplo da FSM-Monitor com propriedades modelado para a Ethernet DM9000A

A propriedade $P1$ descreve que a camada física da DM9000A nunca dever ser ativada se o modo de operação ainda está indefinido. Em outras palavras, nunca pode haver uma transição para o estado *PHYUP* quando o estado *UndefOperMode* é um estado corrente. Como ela é uma propriedade do estado *globalstate*, ela é herdada por todos os estados da FSM-Monitor, uma vez que todo estado filho herda as propriedades de seus estados pai.

A propriedade $P2$ descreve uma propriedade específica do estado *PHYUP*. Ela foi especificada no *PHYUP* uma vez que fora dele esta propriedade não faz sentido ser checada. Esta propriedade define uma regra cujo manual de aplicação de DM9000A deixa bem explicita: *"Nunca defina o valor do tamanho da carga útil do dado escrito na memória da ethernet enquanto um pacote estiver sendo enviado"*. Em outras palavras, os estados *SendData* e *SetLen* não podem ser estados correntes simultaneamente.

Como já explicado no Capítulo 4, as propriedades são escritas através de fórmulas LTL e são sintetizadas em autômatos Büchi. Os autômatos Büchi são extensões de autômato finito que suportam entradas infinitas. Autômatos Büchi aceitam como entrada sequências infinitas quando existe uma execução que visita pelo menos um dos estados finais infinitas vezes.

Quando se define uma propriedade, os estados de aceitação dos autômatos Büchi são alcançados quando a propriedade é satisfeita. Entretanto, isto não é interessante para identificar violações. Dessa maneira as propriedades da FSM-Monitor são negadas então são sintetizadas. Logo, os estados de aceitação dos autômatos gerados são alcançados quando a propriedade não é satisfeita.

As transições de estado nessas máquinas Büchi são disparadas na ocorrência dos mesmos eventos que disparam as transições de estados da FSM-Monitor. Essas transições também devem respeitar expressão, entretanto estas expressões são extraídas das formulas LTL durante a síntese dos autômatos Büchi.

Logo deve-se adaptar o sistema de transição para permitir transições de estados nas máquinas Büchi concorrentemente com as transições de estados da FSM-Monitor, além de parar o sistema de transição quando um estado de aceitação for alcançado.

Para isso 5 novos conjuntos serão definidos. O primeiro, L, é o conjunto de todos os

estados dos autômatos Büchi. O conjunto B é o conjunto de todos os estados correntes dos autômatos Büchi, tal que $B \subset L$. O outro conjunto definido é o conjunto F representando todos os estados finais de todas as máquinas de estados Büchi, relembrando que estes estados finais representam violações das propriedades, e C representa os estados finais das assertivas do tipo *CRITICAL*, tal que $C \subset F$. Por fim, tem-se o conjunto $G = P \cup B$, ou seja, G é o conjunto de todos os estados correntes na execução da FSM-Monitor.

Assim como as transições entre estados da FSM-Monitor, as transições entre estados dos autômatos Büchi também devem respeitar uma expressão f extraída a partir das formulas LTL. Dessa maneira, pode-se então definir o sistema de transição final como:

$$(G, M) \xRightarrow[\tau]{e \bigvee f} (G', M') \tag{5.16}$$

É importante também definir uma situação na qual a FSM-Monitor deve se encontrar quando houver uma violação de alguma propriedade. Para isso, foi definido um estado $q_r \in Q$ que representa o resto de rejeição. Dessa maneira, toda FSM-Monitor contém q_r, sendo ele o estado alvo de transições quando houver uma violação de alguma propriedade representada pelas máquinas Büchi.

Com base nestas definições é possível definir as transições de estados na FSM-Monitor concorrentemente com as checagens de propriedades. Inicialmente pode-se ver nas equações 5.17 e 5.19 transições sem e com violação, respectivamente. É interessante ressaltar que a equação 5.16 será destrinchada detalhadamente em 2 passos, porém em um único evento.

$$((\{g\}\dot{\cup}B) \cup (\{p\}\dot{\cup}P), M) \overset{f}{\underset{\tau}{\Rightarrow}} (\{g'\} \cup B \cup (\{p\}\dot{\cup}P), M) \overset{e}{\underset{\tau}{\Rightarrow}} (\{g', p'\} \cup G, M') \iff \begin{cases} (i) g \in L, g' \in L, p \in Q, p' \in Q \\ (ii)\ g \notin F, g' \notin F \\ (iii) \forall g_i \in B \mid \nexists g_i \in F \end{cases} \tag{5.17}$$

A equações 5.17 mostra que quando há uma transição sem violação do autômato Büchi, a transição de estados da FSM-Monitor ocorre de acordo com o previamente definido nas regras de transição. Porém, quando há uma violação e esta violação não é crítica (*WARNING*), há transições para os estados finais das máquina *Büchi* que representam as propriedades violadas, porém não há nenhum tipo de efeito colateral na execução da FSM-Monitor, uma vez que uma violação deste tipo de propriedade não influencia na execução do dispositivo e nem o leva a um estado indefinidos. A equação (5.18) formaliza

a ocorrência deste tipo de violação.

$$((\{g\}\dot{\cup}B) \cup (\{p\}\dot{\cup}P), M) \overset{f}{\underset{\tau}{\Rightarrow}} (\{g'\} \cup B \cup (\{p\}\dot{\cup}P), M) \overset{e}{\underset{\tau}{\Rightarrow}} (\{g', p'\} \cup G, M') \iff \begin{cases} (i) g \in L, g' \in L, p \in Q, p' \in Q \\ (ii)\ g \notin F, g' \in F \\ (iii) \forall g_i \in B \mid \exists g_i \in F \end{cases} \tag{5.18}$$

Quando há uma violação de uma propriedade crítica (*CRITICAL*), haverá uma transição de estados da FSM-Monitor diretamente para o estado de rejeição q_r, como pode ser visto na equação (5.19).

$$((\{g\}\dot{\cup}B) \cup (p\dot{\cup}P), M) \overset{f}{\underset{\tau}{\Rightarrow}} (\{g'\} \cup B \cup (\{p\}\dot{\cup}P), M) \overset{e}{\underset{\tau}{\Rightarrow}} (\{g', q_r\} \cup G, M') \iff \begin{cases} (i) g \in L, g' \in L, p \in Q \\ (ii)\ g \notin F, g' \in C \end{cases} \tag{5.19}$$

Para a equação 5.17, as regras (ii) e (iii) mostram que, quando não há uma transição para um estado final em nenhum autômato Büchi, não há violação de propriedades e, por isso, se a expressão e for verdadeira, haverá uma transição normal na FSM-Monitor.

Caso contrário, se houve pelo menos uma transição para um estado final de qualquer máquina de estados Büchi, haverá obrigatoriamente uma transição para o estado de rejeição q_r na FSM-Monitor.

Com base na semântica operacional da HFSM-S, o capítulo a seguir irá descrever a linguagem TDevC e a sua semântica.

5.2 Resumo

Este capítulo apresentou a FSM-Monitor com detalhes. Foi detalhado cada elemento da máquina de estados e a sua semântica operacional. A semântica operacional da FSM-Monitor detalha como cada transição de estados é formalmente realizada durante a execução do MDDC.

6 A LINGUAGEM TDEVC

Como dito no início deste capítulo, a primeira etapa do fluxo proposto é a especificação *TDevC* das características estruturais de um dispositivo e características comportamentais ideais do seu respectivo *device driver*. Esta linguagem vem sendo desenvolvida desde 2009 e vem sofrendo várias adaptações, como podem ser acompanhadas através dos trabalhos (MACIEIRA; LISBOA; BARROS, 2011; MACIEIRA; BARROS; ASCENDINA, 2014), até chegar à atual sintaxe e ao atual poder de expressão.

A linguagem TDevC permite especificar elementos estruturais, protocolos de acesso a memória de dispositivos e comportamentos ideais para o uso destes periféricos. Este último, como será explicado nos próximos parágrafos, é representado através de uma FSM-Monitor.

Para uma melhor organização desta seção ela será dividida em duas seções: Definição da Interface de Comunicação (Seção 6.1) e Definição do Protocolo de Comunicação (Seção 6.2). Todos os exemplos da sintaxe da linguagem utilizados nas explicações foram retirados de uma especificação real de um controlador *Ethernet DM9000A*.

O controlador da Ethernet DM9000A (DAVICOM, 2005; DAVICOM, 2006) é um dispositivo de rede cabeada que implementa parte da pilha de protocolos do modelo iso / osi (ZIMMERMANN, 1980).

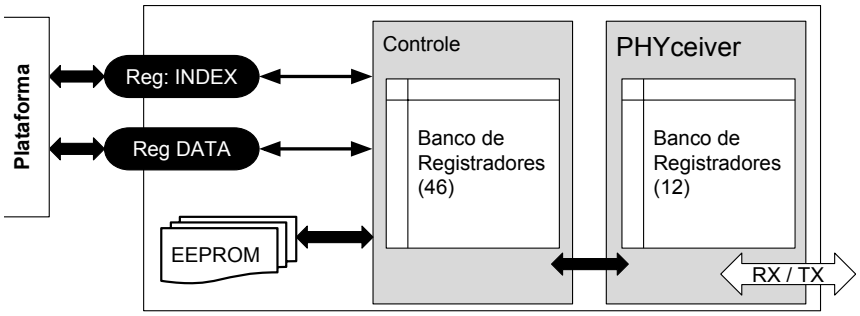


Figura 47 – Diagrama de blocos simplificado da Ethernet DM9000A

A Figura 47 mostra o diagrama de blocos simplificado da arquitetura do DM9000A. Neste diagrama é possível ver que este dispositivo conta com 2 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, como o próprio nome diz, é responsável pelo controle de todas as funcionalidades que dizem respeito às camada acima da camada física implementadas pela DM9000A. Já a unidade de *PHYceiver* é responsável pelo controle de todos as funcionalidade respectivas à camada física do protocolo de rede.

Para que seja possível seu controle a partir de uma cpu, este dispositivo conta com uma interface de comunicação contendo apenas 2 registradores de 8 bits cada: *INDEX* e

DATA. Estes registradores são a porta de entrada para o acesso ao banco de registradores da unidade de controle.

Cada registrador interno, ou seja, dos banco de registradores da unidade de controle e unidade *PHYceiver*, é identificado através de um endereço. Logo, para eles serem acessados externamente a cpu faz uso dos 2 registradores da interface do dispositivo, informando o endereço do registrador interno no registrador *INDEX* e o dado a ser 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 externos da interface. Para efetuar qualquer leitura e escrita neles é precisa usar registradores específicos no banco de registradores da unidade de controle.

Devido a pilha de protocolos implementada pelo DM9000A, este dispositivo se mostrou completo para ser utilizado como exemplo tanto na explicação da linguagem *TDevC* quanto para parte dos experimentos, onde, neste último, outro dispositivos também foram utilizados.

6.1 Sintaxe da Seção da Interface de Comunicação TDevC

A definição da interface de comunicação da linguagem *TDevC* é composta pela especificação dos registradores do dispositivo, os formatos destes registradores e padrões de dados através das construções *register*, *format* e *pattern*, respectivamente. A Figura 48 mostra um exemplo dessas construções.

```
device (dm9000a) {
    pattern RXNOERROR = mask(xxxx0000);

    format physicalAddrfmt {
        RW PAB[7:0];
    }

    external register indexReg(0x00) alias INDEXREG {
        RW INDEX [15:0];
    }

    internal IntRegsProt register networkStatusReg(0x00) alias NSR{
        READ SPEED [7];
        READ LINKST [6];
        RW WAKEST [5];
        reserved [4];
        RW TX2END [3];
        RW TX1END [2];
        READ RXOV [1];
        reserved [0];
    }

    internal IntRegsProt register phyAddrReg5(0x15) alias PAR5 =
        physicalAddrfmt;
...
```

Figura 48 – Exemplo da descrição da interface de comunicação na linguagem TDevC

Primeiramente, toda especificação TDevC começa com a palavra reservada **device** seguida pelo nome do dispositivo cujo modelo está sendo especificado, como mostrado na linha um da Figura 48.

A linha 2 da Figura 48 mostra um exemplo de uma declaração de um padrão, através da palavra reservada pattern. O uso de padrões permite especificar formatos numéricos de máscaras. Eles tornam a descrição dos comportamentos mais claras e menos propensa a erros. Este tipo de especificação lembra bastante a declaração de variáveis constantes de linguagens de programação, mas além de valores fixos, pode-se especificar também formatos fixos de dados. o padrão RXNOERROR especificado na Figura 48-linha 2 define o formado do dado que deve ser lido, independente da origem deste dado, quando houver um erro na transmissão de um pacote de dados via dispositivo *Ethernet DM9000A*. Neste exemplo a palavra reservada *mask* define a máscara que um dado lido ou escrito deve respeitar ao ser comparado com este padrão. A máscara é definida com a composição de valores do tipo *don't care* (x), tipo zero (0) ou tipo um (1). Valores do tipo *don't care* indicam que o bit naquela determinada posição pode ser valorado com zero (0) ou um (1). Para o tipo zero, é obrigatório que o bit em questão tenha o valor zero (0), e para o valor um (1), que o bit em questão tenha o valor um (1). Neste caso, o dado deve conter obrigatoriamente os últimos quatro bits iguais ao valor zero (0);

A palavra reservada *register*, como o nome mesmo diz, representa a declaração de um registrador real de um dispositivo. Os registradores podem ser declarados explicitamente com seus nomes, suas visibilidades, seus apelidos ou nomes fantasias (*alias*), seus campos e permissões de acesso e seus endereços físicos, como mostrado na Figura 48-linhas 8 a 20 ou podem ser declarados utilizando formatos de registradores previamente declarados através da palavra reservada *format*, como mostram as linhas 4, 5 e 6 da Figura 48.

A declaração dos formatos de registradores é muito parecida com a declaração dos próprios registradores, diferenciando apenas no fato de que os formatos não são vinculados a nenhuma visibilidade, nenhum endereçamento e nome fantasia, uma vez que esses atributos somente fazem sentido quando atrelados a registradores reais.

Como pode ser visto na Figura 48-linhas 8 e 12, existem dois tipos de visibilidade de registradores: externos, representados através da palavra reservada **external**, e internos, representados através da palavra reservada **internal**. Os registradores externos são todos aqueles que são mapeados na faixa de endereçamento da plataforma, ou seja, os registradores que são diretamente acessados pelos componentes mestres do sistema durante a comunicação. Usando como exemplo a DM9000A, este registradores seriam o *INDEX* e o *DATA*.

Já os registradores internos, são aqueles que não são endereçados diretamente na faixa de endereços do sistema. Estes são acessados através de um protocolo de acesso aos registradores externos, comumente implementado nas camadas de software dependente de hardware. Na Figura 48-linhas 12 e 22 tem-se exemplos de dois registradores internos que

utilizam um protocolo de acesso chamado *IntRegsProt*. A declaração e especificação dos protocolos serão detalhadas na Seção 6.2, a seguir.

É importante destacar que os endereços atribuídos aos registradores são endereços absolutos dentro da faixa predefinida para os dispositivos, porém, para os registradores externos, eles são relativos no escopo da plataforma. Tomando como exemplo o registrador externo *indexReg*, pode-se perceber que ele é o registrador de endereço *0x00* no dispositivo. Entretanto, caso o periférico em questão se encontre na faixa de endereço *0x00A-0x00E* do sistema, seu endereço relativo na faixa de endereçamento do sistema referente a este dispositivo é dado por *0x00* e traduzido, durante a síntese pela *TDevCGen*, para o endereço absoluto *0x00A* na plataforma.

Ainda em relação ao endereçamento dos registradores, registradores externos estão em um escopo de endereçamento diferente dos registradores internos, uma vez que os externos têm seus endereços vinculados e traduzidos diretamente na plataforma alvo e os internos são relativos ao protocolo de acesso. Assim, com esta relação dos registradores internos com os seus protocolos, fica mais evidente que cada protocolo definido carrega consigo um escopo diferenciado de endereçamentos, permitindo que registradores internos com diferentes protocolos e registradores externos compartilhem na especificação o mesmo valor numérico de endereços. Um exemplo disto pode ser visto na Figura 48-linhas 8 e 12.

O atributo *alias* define um nome fantasia ou apelido para o registrador, podendo ser usados na especificação do protocolo na linguagem. Como mencionado, o *alias* não é obrigatório; o objetivo deste mecanismo é apenas permitir a referência simplificada do registrador na especificação, ao mesmo tempo mantendo a sua descrição completa e clara. Normalmente os *datasheets* (documentações oficiais de dispositivos) fazem referência ao nome completo do registrador em sua apresentação e referência de um nome fantasia, comumente suas iniciais, na sua aplicação no uso do dispositivo.

Os campos dos registradores, Figura 48-linhas 5, 9 e 13 a 20, são subdivisões lógicas descritas nos *datasheets* dos dispositivos. Normalmente cada subdivisão tem uma função específica. Um valor atribuído a um campo pode acarretar em um comportamento no dispositivo. Logo, para tornar mais clara a descrição em TDevC e reduzir a possibilidade de erros na comparação de valor de registradores, a linguagem permite especificar campos aninhados. Registradores contêm campos, seus campos por sua vez podem também conter subcampos e assim por diante, sem limite de níveis de aninhamento.

Se o *datasheet* informa que o campo é reservado, ou seja, de uso interno, não estável ou ainda não foi definido pelo fabricante, usa-se a declaração de campo reservado, como mostram as linhas 16 e 20 da Figura 48. Percebe-se que na declaração de um campo reservado não se especifica nem permissão de acesso, uma vez que a única permissão de acesso aceitável (mas não recomendada) é somente leitura, e nem subcampos, uma vez que subcampos de um campo reservado também serão reservados.

A outra maneira de definir campos é usada quando estes são campos válidos para uso.

Mostrada nas linha 5, 9, 13, 14, 15, 17, 18 e 19 da Figura 48, este tipo de declaração requer três atributos obrigatórios e permite dois atributos opcionais. Começando pelos obrigatórios, o primeiro atributo é a permissão de acesso. Os campos podem ter acessos somente para escrita (*WRITE*), somente para leitura (*READ*) ou para escrita e leitura (*RW*).

A seguinte atributo obrigatório é o nome do campo. Todo campo deve conter um nome, mesmo que este tenha subcampos. Estes nomes são usados para se ter uma referência destes campos na definição do protocolo da linguagem *TDevC*. Os campos de um registrador são referenciados através do nome ou *alias* do registrador seguido de um *"."* acompanhado do nome do campo. O mesmo padrão serve para subcampos de campos e assim por diante. Um exemplo pode ser visto na sintaxe a seguir que declara um campo *INDEX* do registrador *indexReg*.

```
indexReg.INDEX
```

Se, por exemplo, o campo *INDEX* do registrador *indexReg* tivesse um subcampo chamado *INDEXHI*, sua referência seria feita da seguinte maneira:

```
indexReg.INDEX.INDEXHI
```

e assim por diante.

6.2 Sintaxe da Seção do Protocolo de Comunicação da TDevC

Uma definição do protocolo de comunicação em *TDevC* é composta por construções que usam sintaxes específicas para declarações de protocolos de acesso a registradores internos, declarações de variáveis de contexto e a declaração da FSM-Monitor, suas atribuições de dados e declarações de suas propriedades comportamentais. Para simplificar a explicação e não deixá-la confusa, este documento irá abordar na sequencia cada uma destas construções, iniciando com as declarações de protocolos, seguindo com as declarações de variáveis e da FSM-Monitor, finalizando com as declarações das propriedades comportamentais.

Como explicado na seção anterior, os protocolos são utilizados para definir os procedimentos de acesso a registradores internos, partindo de acessos a registradores externos, direta ou indiretamente. As linhas 1 e 12 da Figura 49 mostram dois exemplos de especificação de protocolo, também utilizando como base uma especificação do controlador *Ethernet DM9000A*, para se ter acesso aos registradores tanto da unidade de controle quanto do módulo *PHYceiver*. A Figura 50 mostra no diagrama de blocos da Ethernet onde estes dois protocolos estão sendo usados.

O bloco de protocolo é inicializado pela palavra reservada *mapping* seguida por um nome identificador. Dentro do bloco é possível especificar o registrador ou campo de registrador que representará o endereço e o dado do registrador interno associado ao protocolo através das palavras reservadas *address* e *data*, respectivamente. Através da Figura 49-linhas 2, 3, 13 e 14 é possível ver exemplos de definição dos registradores ou

```
1  mapping IntRegsProt {
       address: INDEXREG(0X00);
3      data: DATAREG;
       readingtrigger{
5          read(DATAREG);
       }
7      writingtrigger{
           write{DATAREG};
9      }
   }
11
   mapping ProtPHYRegs{
13     address: EPAR.REGADDR(0X00);
       data: {EPDR.EE_PHY_H;EPDR.EE_PHY_L}
15     readingtrigger{
           write(EPCR) = 0x0C;
17     }
       writingtrigger{
19         write(EPCR) = 0x0A;
       }
21 }
```

Figura 49 – Exemplo de declaração de protocolos de acesso a registradores internos na linguagem TDevC

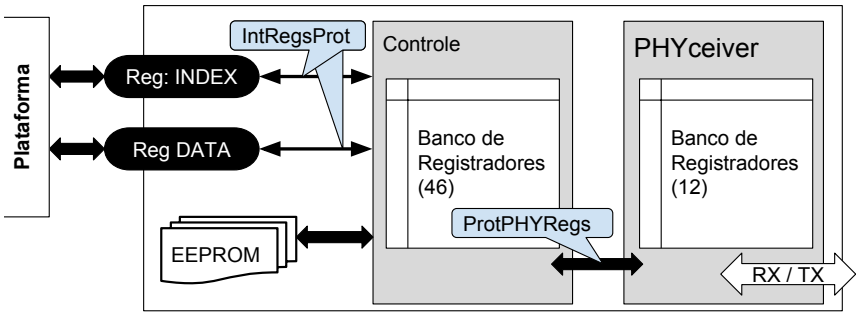


Figura 50 – Diagrama de blocos simplificado da Ethernet DM9000A com referência aos protocolos entre bancos de registradores

campos de registradores utilizados para definir endereços e dados de registradores internos associados aos protocolos. Na Figura 49-linhas 2 e 3, respectivamente, por exemplo, o protocolo *IntRegsProt* define que todo registrador interno associado será acessado sempre que o seu endereço for definido no registrador *INDEXREG* e houver uma leitura ou escrita de um valor no registrador *DATAREG*.

O mesmo pode ser observado na Figura 49-linhas 13 e 14, entretanto o campo de registrador que define o endereço de todos os registradores associados ao protocolo *ProtPHYRegs* é o *EPAR.REGADDR* e os campos de registrador que definem os dados lidos ou escritos dos registradores associados a este protocolo são *EPDR.EE_PHY_H* e *EPDR.EE_PHY_L*, porém concatenados de maneira com que *EPDR.EE_PHY_H* seja a parte mais significativa do dado e *EPDR.EE_PHY_L* a menos significativa.

É preciso informar o gatilho que indica quando o dado está pronto para ser lido ou escrito. Para isso as construções *readingtrigger writingtrigger* foram definidas. Usando o exemplo da Figura 49-linhas 15 a 20, a especificação do Ethernet DM9000A diz que para

ler e escrever um dado no banco de registradores do módulo *PHYceiver*, é preciso escrever os valores *0x0C* e *0x0A*, respectivamente, no registrador *EPCR* informando que o endereço e o dado já foram configurados para realizar a leitura ou escrita.

É importante deixar claro que a utilização da construção *protocol* tem como objetivo simplificar a descrição em *TDevC*, tornando-a mais clara quando ao acesso a registradores internos.

```
1  var t1 = 1;
   var rxlowlen;
3  var pkgcounter;
   var rxlen;
5
   globalstate {
7      orthoregion ethOperationMode {
           initialstate UNDEF_OPER_MODE {
9              addexitpoint(OPER16BITS){
                   READ(ISR.IOMODE) == 0
11             }

13             addexitpoint(OPER8BITS){
                   READ(ISR.IOMODE) == 1
15             }
               addentrypoint {
17                 WRITE(GPR.RST) == 1
               }
19         }
           state OPER16BITS {
21             addexitpoint(OPER8BITS){
                   READ(ISR.IOMODE) == 1
23             }
           }
25         state OPER8BITS {
               addexitpoint(OPER16BITS){
27                 READ(ISR.IOMODE) == 0
               }
29         }
       }
31 ...
```

Figura 51 – Exemplo da descrição de um protocolo de comunicação da linguagem TDevC

A declaração de variáveis e da FSM-Monitor, como podem ser vistas no exemplo na Figura 51, utilizam as palavras reservadas *var* e *globalstate*, respectivamente.

As variáveis são utilizadas quando se quer adicionar algum contexto durante validação, diferentemente de versões anteriores da linguagem onde a validação não tinha contexto e nem noção de estados de execução, sendo completamente *stateless*. Toda variável tem um escopo global e as atribuições feitas a elas ocorrem durante as transições da FSM-Monitor. As linhas de 1 a 4 da Figura 51 mostram quatro exemplos de declaração de variáveis. Na Figura 51-linhas 1 é mostrada exatamente uma declaração de variável com atribuição de um valor inicial.

Com todas as variáveis já declaradas, pode-se então declarar a FSM-Monitor. A Figura 52 mostra graficamente a máquina de estados especificada na Figura 51. Como mencionado anteriormente, inicia-se a especificação da FSM-Monitorcom a palavra reservada *globalstate*

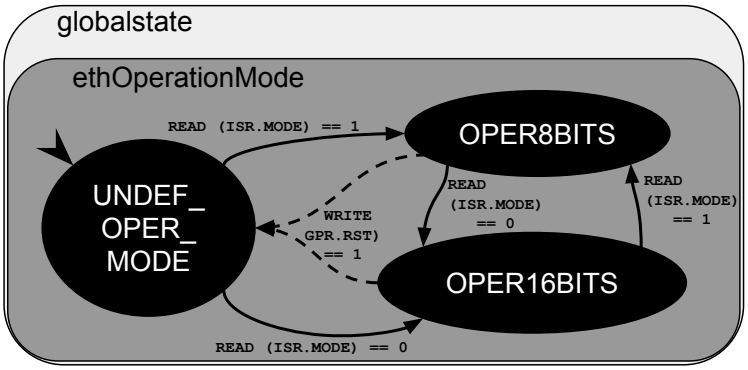


Figura 52 – Representação gráfica da máquina de estados especificada na Figura 51

Como a própria palavra reservada usada já deixa claro, *globalstate* indica o estado global, único e pai de todos os estados seguintes, no primeiro nível (nível raiz) da FSM-Monitor. Assim como um estado qualquer da máquina hierárquica, o estado global é segmentado em regiões ortogonais através de sub-blocos iniciados com a palavra reservada *orthoregion*, respeitando a definição da FSM-Monitordescrita na Seção 5. A Figura 51-linha 6 mostra o início do bloco do estado global e, consequentemente, da FSM-Monitor.

Cada região ortogonal representa linhas independentes de execução dentro de um estado. Assim, sua sintaxe contém também a declaração de sub estados, permitindo então a declaração de uma máquina hierárquica Toda região ortogonal tem obrigatoriamente um único estado inicial declarado através da sintaxe de estados iniciada pela palavra reservada *initialstate*. Outros estados são opcionais, porém, de acordo com a definição da máquina hierárquica, todos os demais estados são intermediários, declarados através da sintaxe de estados iniciada pela palavra reservada *state*. Dessa maneira, essa estrutura sintática cobre todos os tipos de estados definidos no Seção 5: estado global, estado inicial e estado intermediário.

O exemplo mostrado na Figura 51 contém uma região ortogonal chamada de *ethOperationMode*, localizada na linha 7. O seu estado inicial, na linha 8, é o *UNDEF_OPER_MODE* e seus 2 estados intermediários estão declarados nas linhas 20 e 25 e se chamam respectivamente *OPER16BITS* e *OPER8BITS*. Esta região ortogonal representa a seleção do modo de operação da Ethernet DM9000A. Ele é inicialmente desconhecido pelo *device driver* e de acordo com a seleção passa a ser de 8 bits ou 16 bits. Na explicação seguinte sobre a sintaxe de estados será detalhado como ocorre essa seleção e como são disparadas as transições para os estados dos modos de operação.

A única diferença que existe entre a declaração de um estado inicial e um estado intermediário é apenas a palavra reservada que inicia o bloco: *initialstate* e *state* que, respectivamente.

Como pode ser visto na Figura 51-linhas 9, 13, 16, 21 e 26, dentro de cada bloco de definição de estado dois atributos são responsáveis por especificar as transições entre estados: *exitpoints* and *entrypoints*. O blocos *exitpoints* definem as transições de saída,

sempre especificados nos estados que originam a transição, sempre disparadas pelas eventos e respeitando a veracidade das expressões lógicas, as quais foram formalizadas no capítulo anterior. As transições representadas pelos *exitpoints* pode ser vistas graficamente na Figura 52 através das setas não tracejadas. Nos exemplo da Figura 51 é possível observar quatro exemplos de transições definidas através de *exitpoints* (linhas 9, 13, 21 e 26).

Como dito anteriormente, inicialmente o *driver* da Ethernet DM9000A é inicializado sem saber em qual modo de operação o dispositivo se encontra. Somente após perguntá-lo sobre esta informação é que o *driver* terá ciência deste modo de operação e poderá transmitir e receber informações de maneira correta.

Dessa maneira, o estado inicial *UNDEF_OPER_MODE* especifica dois *exitpoints*: *addexitpoint(OPER16BITS)* e *addexitpoint(OPER8BITS)*. Estas duas transições são disparadas quando, respectivamente, há uma leitura do registrador *ISR.IOMODE* e o seu conteúdo é 0 (zero) ou há uma leitura do registrador *ISR.IOMODE* e o seu conteúdo é 1 (um), e elas tem como estados alvo o estado *OPER16BITS* e o estado *OPER8BITS*. Resumindo: quando o *device driver* requisita ao dispositivo informações sobre o modo de operação, dependendo do conteúdo destas informações, o monitor irá para o estado de 8-bits ou 16-bits.

A construção *entrypoint*, mostrada na Figura 51, linhas 16, e representada graficamente na Figura 52 através das setas tracejadas, foi criada como um açúcar sintático. Durante a síntese todas essas construções são convertidas em *exitpoints* partindo de todos os outros estados da mesma região ortogonal cujo o estado pertence e terão como alvo este mesmo estado que definiu o *entrypoint*. A ideia é justamente facilitar e transparecer a ideia de transições comuns a todos os estados de sub-máquina hierárquica. Isso é muito comum, por exemplo, em situações de *reset*. Quando um *reset* de software é disparado, independente do ponto do protocolo em que a execução da comunicação esteja, ela deve retornar a um estado inicial ou de *reset*.

A construção *addproperty*, observada na sintaxe do bloco de definição de estados e exemplificada na Figura 53, é utilizada para especificar as propriedades que serão validadas durante a execução do sistema. Como é possível ver, existem dois tipos de propriedades: *critical*, para expressar propriedades críticas da execução e *warning*, para propriedades informativas e que não causam danos direto ao protocolo e à comunicação entre os periféricos. As propriedades críticas, quando violadas interrompem imediatamente o monitoramento, uma vez que não há mais nenhuma garantia sobre o estado no qual o dispositivo se encontra no momento após a violação.

Atualmente as propriedades são especificadas unicamente através de asserções descritas em fórmulas ltl. Na Figura 53-linhas 1 a 3 mostram a especificação de uma propriedade atribuída ao estado chamado *PHYUP* da Ethernet DM9000A.

Este estado é alcançado pela FSM-Monitorquando o *driver* inicializa a unidade *PHY-ceiver*, tornando a *Ethernet* pronta para transmitir e receber pacotes de rede.

```
1  addproperty(critical) UndefinedOperMode{
       ltlf([](~UNDEF_OPER_MODE))
3  }
   addproperty(critical) WriteBeforeLen{
5      ltlf(~TXWRPKGLEN U TXWRPKGWR)
   }
7  addproperty(critical) LenBeforeSend{x
       ltlf(~TXSDPKGSDING U TXWRPKGLEN)
9  }
   addproperty(warning) NeverLenAndSend {
11     ltlf([]( ~(TXSDPKGSDING && TXWRPKGLEN)))
   }
```

Figura 53 – Exemplo da declaração de propriedades em estados na linguagem TDevC

Como mencionado, o dispositivo DM9000A não pode operar sem que o *driver* saiba em qual modo de operação o dispositivo se encontra. Portanto, a propriedade descrita na linha 2 indica que, quando a execução estiver neste ponto do protocolo, sempre (operador temporal *always* "[]") *UNDEF_OPER_MODE* deve ser negado (operador lógico "~"), ou seja, este estado não pode ser um estado corrente em nenhuma região ortogonal durante execução da FSM-Monitor. Traduzindo para linguagem natural: Nunca o dispositivo DM9000A pode ter a camada física ligada sem que o *driver* tenha conhecimento que o modo de operação do dispositivo se encontra definido.

Continuando o exemplo demostrado na Figura 53 e assumindo que os estados TXWRPKGLEN, TXWRPKGWR e TXSDPKGSDING foram previamente declarados, a propriedade "~TXWRPKGLEN U TXWRPKGWR" significa que "o dispositivo DM9000A não deve ir para o estado de definição do tamanho do dado antes ter passado pelo estado de escrita de algo na memória do dispositivo". Já a propriedade "~TXSDPKGSDING U TXWRPKGLEN" significa que "o dispositivo DM9000A não deve ir para o estado de envio de dado antes de ter passado pelo estado de definição do tamanho do dado escrito em memória". Por fim, a propriedade "[](~(TXSDPKGSDING TXWRPKGLEN))" significa que "sempre no dispositivo DM9000A os estados de envio de dados e de definição do tamanho de dados escritos na memória do dispositivo não podem estar concorrentemente em execução no mesmo momento".

Com o conhecimento da sintaxe da linguagem consolidado, a semântica será apresentada na seção seguinte.

6.3 Semântica da TDevC

Como já destacado anteriormente, a TDevC é uma linguagem de especificação. Diferentemente de uma linguagem de programação, a TDevC tem como principal função servir como o *front-end* da abordagem, permitindo que todos os elementos de uma FSM-Monitor possam ser especificados.

Dessa maneira, a semântica da TDevC, ou seja, o sentido dos elementos da sintaxe e a interpretação de suas construções deve seguir uma lógica de armazenamento e montagem

de um modelo, ao invés de uma sequência lógica de execução, como se dá a semântica de uma linguagem de programação.

Como pilar da semântica da TDevC, existe uma memória onde cada elemento da linguagem será armazenado e vinculado a outros elementos, formando um contexto e um sentido para a especificação. Para isto, define-se formalmente uma memória **mem**, tal que é possível armazenar qualquer elemento (independente da sua complexidade) da TDevC e associá-lo a um identificador.

Para acessar esta memória serão definidos duas funções: **mem_sto** e **mem_ret**, tal que:

$$mem\_sto : ident \times ref \rightarrow ident \tag{6.1}$$

$$mem\_ret : ident \rightarrow ref \tag{6.2}$$

A função **mem_sto** tem como objetivo persistir um elemento da TDevC na memória *mem*. Esta função tem como domínio a tupla $\langle ident, ref \rangle$ contendo respectivamente um identificador e a estrutura de um elemento da TDevC que será persistido na memória. Assim, o elemento *ref* gravado é associado a um identificador *ident*. Esta função tem como contradomínio o conjunto dos identificadores dos elementos persistidos.

Já a função **mem_ret** tem como objetivo recuperar o elemento da memória *mem* a partir de seu identificador. Esta função tem como domínio o conjunto dos identificadores dos elementos gravados na memória *mem* e tem como contradomínio o conjunto dos elementos da TDevC persistidos na memória *mem*.

Cada registro na memória *mem* é composta por uma tupla ⟨chave, valor⟩, onde as chaves compõe um conjunto de elementos e, por definição, não pode haver repetição de elementos.

Para exemplificar o uso dessas funções, suponha que três persistências sequenciais são realizadas na memória através da função mem_sto. São elas:

1. $mem\_sto(A, 1) \rightarrow A$
2. $mem\_sto(B, x) \rightarrow B$
3. $mem\_sto(A, 2) \rightarrow A$

Através da Figura 54 é possível ver o estado da memória em três instantes diferentes, sempre após uma das escritas ser realizada. É possível perceber que nos estado de tempo que representam a memória *mem* e *mem'* os elementos A e B são registrados, representando os valores *1* e x. É possível perceber também que n momento representando o estado da memória *mem"*, após uma escrita de um elemento já existente, apenas o valor *1* do elemento em questão é substituído pelo valor *2*, uma vez que não pode haver identificadores repetidos na memória.

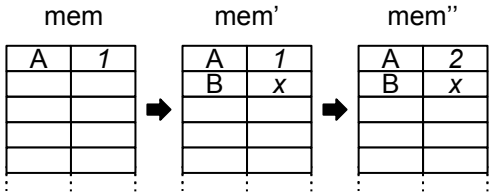


Figura 54 – Exemplo do estado da memória ao longo do tempo com base no uso da função *mem_sto*

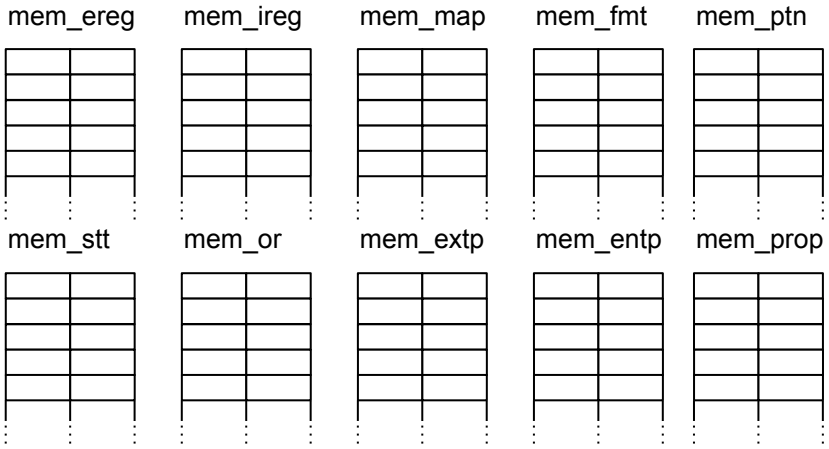


Figura 55 – Representação dos bancos de memória simbólicos utilizados pela TDevCGen

Por questões de simplificação da semântica, esta abordagem dividiu a memória em diferentes bancos, cada um deles sendo responsável por armazenar um único tipo específico de elemento da TDevC. A Figura 55 mostra esta divisão. São elas:

- **mem_ereg**: Banco de memória que irá armazenar os elementos do tipo *External Registers*
- **mem_ireg**: Banco de memória que irá armazenar os elementos do tipo *Internal Registers*
- **mem_map**: Banco de memória que irá armazenar os elementos do tipo *Mapping*
- **mem_fmt**: Banco de memória que irá armazenar os elementos do tipo *Format*
- **mem_ptn**: Banco de memória que irá armazenar os elementos do tipo *Pattern*
- **mem_stt**: Banco de memória que irá armazenar os elementos do tipo *State*
- **mem_or**: Banco de memória que irá armazenar os elementos do tipo *Orthogonal Region*
- **mem_extp**: Banco de memória que irá armazenar os elementos do tipo *Exit Point*
- **mem_entp**: Banco de memória que irá armazenar os elementos do tipo *Entry Point*

- **mem_prop**: Banco de memória que irá armazenar os elementos do tipo *Property*

Segundo essa lógica de divisão da memória *mem* em bancos, houve também a criação de funções específicas, especializando a funções mem_sto e mem_rec, para acessar cada um dos bancos de memória. São elas:

- **mem_ereg_sto e mem_ereg_ret**: Função para persistir e recuperar elementos do tipo *External Registers*
- **mem_ireg_sto e mem_ireg_rec**: Função para persistir e recuperar elementos do tipo *Internal Registers*
- **mem_map_sto e mem_map_rec**: Função para persistir e recuperar elementos do tipo *Mapping*
- **mem_fmt_sto e mem_fmt_rec**: Função para persistir e recuperar elementos do tipo *Format*
- **mem_ptn_sto e mem_ptn_rec**: Função para persistir e recuperar elementos do tipo *Pattern*
- **mem_stt_sto e mem_stt_rec**: Função para persistir e recuperar elementos do tipo *State*
- **mem_or_sto e mem_or_rec**: Função para persistir e recuperar elementos do tipo *Orthogonal Region*
- **mem_extp_sto e mem_extp_rec**: Função para persistir e recuperar elementos do tipo *Exit Point*
- **mem_entp_sto e mem_entp_rec**: Função para persistir e recuperar elementos do tipo *Entry Point*
- **mem_prop_sto e mem_prop_rec**: Função para persistir e recuperar elementos do tipo *Property*

Estas funções tem como domínio e contradomínio subconjuntos do domínio e contradomínio, respectivamente, das funções *mem_sto* e *mem_rec*. Este subconjuntos são os conjuntos que contém os *ident* e as tuplas $\langle ident, ref \rangle$ referentes aos elementos TDevC correspondentes ao banco de memória a qual a função é relacionada. Por exemplo, o domínio da função mem_ereg_sto é o conjunto das tuplas $\langle ident, ref \rangle$, tal que *ref* representa a estrutura de um elemento *External Register* da TDevC. Logo o identificador *ident* deste elemento estará apenas no banco de memória referente aos elementos do tipo *External Register*, fazendo com que a função *mem_ereg_rec* siga o mesmo princípio, onde

o seu domínio é o conjunto de identificadores pertencentes àquele banco de memória (indiretamente associado ao tipo do elemento TDevC)

Outro conjunto de funções de apoio foram definidas para suportar a descrição da semântica da linguagem. Estas funções têm como objetivo fornecer o conjunto de todos os identificadores persistidos em um banco de memória específico. São elas:

- $mem\_ereg\_set \rightarrow \{ident_0, ..., ident_k\}$: Funções que recupera o conjunto dos k identificadores persistidos no banco de memória mem_ereg.
- $mem\_ireg\_set \rightarrow \{ident_0, ..., ident_k\}$: Funções que recupera o conjunto dos k identificadores persistidos no banco de memória mem_ireg.
- $mem\_ma\_set \rightarrow \{ident_0, ..., ident_k\}$: Funções que recupera o conjunto dos k identificadores persistidos no banco de memória mem_map.
- $mem\_fmt\_set \rightarrow \{ident_0, ..., ident_k\}$: Funções que recupera o conjunto dos k identificadores persistidos no banco de memória mem_fmt.
- $mem\_ptn\_set \rightarrow \{ident_0, ..., ident_k\}$: Funções que recupera o conjunto dos k identificadores persistidos no banco de memória mem_ptn.
- $mem\_stt\_set \rightarrow \{ident_0, ..., ident_k\}$: Funções que recupera o conjunto dos k identificadores persistidos no banco de memória mem_stt.
- $mem\_or\_set \rightarrow \{ident_0, ..., ident_k\}$: Funções que recupera o conjunto dos k identificadores persistidos no banco de memória mem_or.
- $mem\_extp\_set \rightarrow \{ident_0, ..., ident_k\}$: Funções que recupera o conjunto dos k identificadores persistidos no banco de memória mem_extp.
- $mem\_entp\_set \rightarrow \{ident_0, ..., ident_k\}$: Funções que recupera o conjunto dos k identificadores persistidos no banco de memória mem_entp.
- $mem\_prop\_set \rightarrow \{ident_0, ..., ident_k\}$: Funções que recupera o conjunto dos k identificadores persistidos no banco de memória mem_prop.

Estas funções dão suporte base para o entendimento da semântica da linguagem TDevC. Cada construção da linguagem tem um reflexo no estado da memória mem e consequentemente nos seus bancos especializados. Como comentando anteriormente, a TDevC é uma linguagem de especificação e, diferentemente de uma linguagem de programação, não há a execução de um programa ou algoritmo. A interpretação da TDevC é refletida na construção de um modelo de máquina de estados e esta construção se dá na memória mem.

A seções a seguir irão justamente expor a semântica da interpretação da TDevC, confrontando a sua sintaxe e o reflexo desta interpretação com o que está sendo persistido

na memória *mem*. Ao final da interpretação de uma especificação TDevC, a memória *mem* deverá conter o modelo de uma FSM-Monitor semanticamente correta.

6.3.1 Semântica da Construção External Register

Como apresentado na Seção 6.1 deste capítulo, o ambiente de definições de interface da TDevC suporta a especificação de dois tipos de registradores: os externos e os internos.

A sintaxe da declaração de um registrador externo é composta pelas palavras reservadas ***external register*** seguidas pelo identificador do registrador, pelo valor do endereço relativo daquele registrador na plataforma ao qual o dispositivo está inserido, um apelido opcional, onde o mesmo é declarado através da palavra reservada ***alias*** seguida pelo identificador do apelido, e pela lista de campos. Esta construção pode ser vista a seguir:

```
external register <reg_ident>(<addr_int>) 'alias <alias_ident>' {
    <access_type> <f1_ident> [<inital_pos_int>:<final_pos_int>];
    <access_type> <f2_ident> [<inital_pos_int>];
    ...
}
```

Ao encontrar este tipo de construção o sintetizador irá persistir este elemento na memória *mem* associado a um identificador. Antes de definir o passo de cômputo resultante da síntese desta construção, é fundamental definir formalmente a estrutura deste elemento.

Seja um registrador externo representado através da 3-tupla $F = \langle addr, formatName, E \rangle$, onde $addr$ é o valor inteiro referente ao endereço relativo, $formatName$ é o nome do formato de campos, caso o registrador seja especificado através de um, e E é a estrutura complexa que representa os campos do registrador. Como já visto no início deste capítulo, é possível especificar um registrador através de um formato de campos predefinido, conhecido como *Format*. Quando isso acontece, o campo $formatName$ da tupla é populado com o identificador do formato, caso contrário este campo é nulo ou vazio. A semântica para este tipo de declaração de registradores será abordada na seção 6.3.4.

A estrutura dos campos dos registradores é representada através de uma 5-tupla $E = \langle name, type, offset, size, E' \rangle$, onde $name$ é o identificador do campo, $type$ é o tipo de acesso a este campo, $offset$ é a posição inicial do campo, $size$ é o comprimento do campo e E' é a 5-tupla de mesma estrutura que esta apresentada e que referencia o campo seguinte a este.

O campo $size$ é composto pelo valor 1, quando o campo descrito contém apenas a posição inicial $inital\_pos\_int$ do campo ([2], por exemplo). Quando há um valor final $final\_pos\_int$ do campo descrito, o valor de $size$ passa a ser definido através de uma simples operação aritmética de subtração entre valor da posição pela inicial, seguida da adição do valor 1.

Desta maneira, o passo de cômputo que o sintetizador realiza quando encontrar uma construção do tipo *External Register* **sem** especificação de *alias*, com base na sintaxe

apresentada acima, é o seguinte:

$$mem\_ereg\_sto(reg\_ident, F); \tag{6.3}$$

Detalhando F tem-se:

$$F = (addr\_int, NULL, E) \tag{6.4}$$

o campo E representa:

$$E = (f1\_ident, access\_type, inital\_pos\_int, final\_pos\_int - inital\_pos\_int + 1, E') \tag{6.5}$$

e E' seria:

$$E' = (f2\_ident, access\_type, inital\_pos\_int, 1, ()) \tag{6.6}$$

É importante perceber que quando não há um campo seguinte, o elemento do próximo campo na tupla E é representado através de uma elemento sem atributos, vazio ou nulo, como pode ser visto em 6.6 através dos parênteses vazios.

Quando há definição de *alias*, além do passo (6.3), há também o seguinte passo de cômputo:

$$mem\_ereg\_sto(alias\_ident, F); \tag{6.7}$$

Isso acontece porque, no modelo especificado, tanto o nome quanto o *alias* deve referenciar elementos equivalentes e não necessariamente o mesmo elemento. Isto não gera nenhum tipo de inconsistência, uma vez que os elementos não são alterados em memória após a síntese. É importante relembrar que se trata de uma linguagem de especificação e diferentemente de uma linguagem de programação, não há execução de algoritmo e nem de contexto de execução em memória. Há apenas da montagem do modelo em memória que servirá como um guia para a geração do código de um monitor. A estrutura dos elementos da TDevC deve ser inalterável após a sua síntese.

Através do exemplo a seguir é possível ver com detalhes a semântica da construção em questão:

```
external register reg_A(0x00) alias RA {
    RW A[0];
    RW B[1:7];
    READ C[8:9];
}
```

A construção acima reflete nos seguintes passos de cômputo:

1. $mem\_ereg\_sto(\text{"}reg\_A\text{"}, (0, NULL, (\text{"}A\text{"}, RW, 0, 1, (\text{"}B\text{"}, RW, 1, 7, (\text{"}C\text{"}, READ, 8, 2, ())))))$
2. $mem\_ereg\_sto(\text{"}RA\text{"}, (0, NULL, (\text{"}A\text{"}, RW, 0, 1, (\text{"}B\text{"}, RW, 1, 7, (\text{"}C\text{"}, READ, 8, 2, ())))))$

É importante destacar os campos finais, onde há uma referência nula do próximo campo, e a composição dos comprimentos dos campos.

6.3.2 Semântica da Construção Internal Register

Já a sintaxe da declaração de um registrador interno é composta pela palavra reservada ***internal*** seguida pelo identificador do mapeamento do protocolo de acesso ao registrador interno, pela palavra reservada ***register***, pelo valor do endereço absoluto daquele registrador internamente no dispositivo, um apelido opcional, onde o mesmo também é declarado através da palavra reservada ***alias*** seguida pelo identificado do apelido, e pela lista de campos. Esta construção pode ser vista a seguir:

```
internal <mapping_ident> register <reg_ident>(<addr_int>) 'alias <
    alias_ident>' {
    <access_type> <f1_ident> [<inital_pos_int>:<final_pos_int>];
    <access_type> <f2_ident> [<inital_pos_int>];
    ...
}
```

A construção ***Internal Register*** é muito semelhante à construção ***External Register***. A única diferença é a especificação do identificador de um protocolo de mapeamento para acesso ao registrador interno. Dessa maneira, essa seção irá mostrar apenas a diferença entre os tipos de registradores.

Ao encontrar este tipo de construção o sintetizador também irá persistir este elemento na memória *mem* associado a um identificador. Assim, seja um registrador interno representado através de uma 4-tupla $F = \langle addr, mapping, formatName, E \rangle$, onde *addr* é o valor inteiro referente ao endereço relativo, *mapping* é o identificador do protocolo de mapeamento de acesso aos registradores internos, $formatName$ é o nome do formato de campos, caso o registrador seja especificado através de um, e E é a estrutura complexa que representa os campos do registrador, exatamente igual ao definido na seção anterior.

Desta maneira, o passo de cômputo que o sintetizador realiza quando encontrar uma construção do tipo *Internal Register* **sem** especificação de *alias*, com base na sintaxe apresentada acima, é o seguinte:

$$mem\_ireg\_sto(reg\_ident, F); \tag{6.8}$$

Detalhando F tem-se:

$$F = (addr\_int, mapping\_ident, NULL, E) \tag{6.9}$$

Para evitar repetições nesta seção, os campos representados pela tupla E não serão detalhados, uma vez que eles já foram detalhados na seção anterior.

Quando há definição de *alias*, além do passo (6.16), há também o seguinte passo de cômputo:

$$mem\_ireg\_sto(alias\_ident, F); \tag{6.10}$$

Através do exemplo a seguir é possível ver com detalhes a semântica da construção em questão:

```
internal reg_bank1 register reg_B(0x08) alias RB {
    RW A[0];
    RW B[1];
}
```

A construção acima reflete nos seguintes passos de cômputo:

1. $mem\_ireg\_sto(\text{"}reg\_B\text{"}, (8, reg\_bank1, NULL, (\text{"}A\text{"}, RW, 0, 1, (\text{"}B\text{"}, RW, 1, 1, ()))))$
2. $mem\_ireg\_sto(\text{"}RB\text{"}, (8, reg\_bank1, NULL, (\text{"}A\text{"}, RW, 0, 1, (\text{"}B\text{"}, RW, 1, 1, ()))))$

6.3.3 Semântica da Construção Format

Como explicado na seção Seção 6.1, a TDevC possui um açúcar sintático que facilita a especificação de registradores com um conjuntos de campos com o mesmo formato. Esta construção, chamada de *Format*, tem uma sintaxe parecida com a sintaxe das declarações de registradores, diferindo apenas pelo fato de que elas não associadas a endereços e nem a protocolos de mapeamento a registradores internos. Em suma, construções do tipo *Format* são apenas listas de campos com identificadores e que pode ser associadas a registradores.

A sintaxe da declaração de um *Format* é composta pela palavra reservada ***Format***, seguida pelo seu identificador e pela lista de campos. Esta construção pode ser vista a seguir:

```
format <fmt_ident> {
    <access_type> <f1_ident> [<inital_pos_int>:<final_pos_int>];
    <access_type> <f2_ident> [<inital_pos_int>];
    ...
}
```

Como as construções anteriores, ao encontrar este tipo de construção o sintetizador irá persistir este elemento na memória *mem* associado a um identificador. Como um *Format* representa apena um agrupamento de campos e, como vimos anteriormente, um conjunto de subsequentes campos é referenciado através apenas de campo inicial (sendo uma lista encadeada), podemos representar esta construção através de campo inicial F.

Desta maneira, o passo de cômputo que o sintetizador realiza quando encontrar uma construção do tipo *Format*, com base na sintaxe apresentada acima, é o seguinte:

$$mem\_fmt\_sto(fmt\_ident, F); \tag{6.11}$$

onde F é a uma representação de um campo inicial, exatamente igual à definição apresentada na Seção 6.3.2 e na Seção 6.3.2.A seguir será apresentado um exemplo onde é possível ver com detalhes a semântica da construção em questão:

```
format fmt_regMem {
    RW A[0:3];
    RW B[4:7];
}
```

A construção acima reflete nos seguintes passos de cômputo:

1. $mem\_fmt\_sto(\text{"}fmt\_regMem\text{"}, (\text{"}A\text{"}, RW, 0, 4, (\text{"}B\text{"}, RW, 4, 4, ())))$

6.3.4 Semântica da Construção Register com Format

Uma outra maneira de declarar registradores, sejam eles internos ou externos, é através do vínculo a um elemento do tipo *Format*, como mostrado na Seção 6.1.

A sintaxe da declaração de um registrador associado a um *Format* é composta pela declaração de um *External Register* ou *Internal Register*, tendo como diferença a declaração dos campos. Ao invés abrir a seção de declaração de campos, os registradores são vinculados a identificadores de construções do tipo *Format*. As construções *External Register* ou *Internal Register* podem ser vistas a seguir:

```
    external register <reg_ident>(<addr_int>) 'alias <alias_ident>' = <
        fmt_ident>
```

```
    internal <mapping_indent> register <reg_ident>(<addr_int>) 'alias <
        alias_ident>' = <fmt_ident>
```

onde para registradores externos tem-se os seguintes passos de cômputo, sem levar em consideração o *alias*:

$$mem\_ereg\_sto(reg\_ident, F); \tag{6.12}$$

Detalhando F tem-se:

$$F = (addr\_int, fmt\_ident, NULL) \tag{6.13}$$

e para registradores Internos tem-se os seguintes passos de cômputo, sem levar em consideração o *alias*:

$$mem\_ireg\_sto(reg\_ident, F); \tag{6.14}$$

Detalhando F tem-se:

$$F = (addr\_int, mapping\_ident, fmt\_ident, NULL) \tag{6.15}$$

Agora, para os dois casos, o campo da tupla *formatName* é populado e a tupla E, referente aos campos do registrador, se mantém nula ou vazia.

6.3.5 Semântica da Construção Pattern

A TDevC possui outro um açúcar sintático que facilita a especificação de combinações de valores aos quais os conteúdos dos registradores podem ser comparados. Esta construção, chamada de *Pattern*, tem uma sintaxe parecida com sintaxe da declaração de constantes de linguagens de propósito geral, entretanto é possível definir valores que representam um intervalo ou um padrão que respeitam uma máscara.

A sintaxe da declaração de um *Pattern* é composta pela palavra reservada ***Pattern***, seguida pelo seu identificador e pela máscara a qual o identificador está associado. A declaração da máscara é feita através da palavra reservada *mask* seguido pelo padrão binários do valor ou intervalo representado. Esta construção pode ser vista a seguir:

```
pattern <ptn_ident> = mask('<bin_int>')
```

Ao encontrar este tipo de construção o sintetizador também irá persistir este elemento na memória *mem* associado a um identificador. Assim, seja um *Pattern* representado através de uma máscara M, onde esta máscara é representada por uma sequência de símbolos binários ou símbolos que indicam indiferença em qual valor está representado. Estes símbolos são conhecidos como *don't care* (representados através de um **x**).

Os símbolos binários indicam explicitamente o valor numérico que se deseja comparar com valores de registradores. Já os símbolos *don't care* indicam pontos na sequencia binária da máscara onde não importam o seu valor numérico. A combinação de valor numéricos com *don't care* permite a especificação de padrões que representam não apenas valores constantes, mas também intervalos. Por exemplo, tem-se a seguinte mascara: **'000011xx'**. Este mascará representa valores números de representados com 8 bits dentro do intervalo entre 12 e 15. Assim, a comparação de igualdade de um valor de um registrador com este *Pattern* seria verdadeira caso o valor do registrador fosse 12, 13, 14 ou 15.

Desta maneira, o passo de cômputo que o sintetizador realiza quando encontra uma construção do tipo *Pattern*, com base na sintaxe apresentada acima, é o seguinte:

$$mem\_ptn\_sto(ptn\_ident, M); \tag{6.16}$$

A seguir será apresentado um exemplo onde é possível ver com detalhes a semântica da construção em questão:

```
pattern allowedValues = mask('0000110x')
```

A construção acima define um padrão que representa os valores numéricos 12 e 13. Este exemplo sintetizado implica nos seguintes passos de cômputo:

1. $mem\_ptn\_sto(\text{"}allowedValues\text{"}, \text{"}0000110x\text{"})$

Os elementos *Pattern* também permitem a definição direta de valores constantes (sem *don't care*) sem a palavra reservada ***Mask***, como pode ser visto na sintaxe abaixo:

```
pattern <ptn_ident> = <int>
```

Por exemplo:

```
pattern allowedValues = 12
```

o que implica no seguinte passo de cômputo:

1. $mem\_ptn\_sto(\text{"}allowedValues\text{"}, \text{"}00001100\text{"})$

O sintetizador simplesmente converte o valor constante para uma representação binária e a armazena do mesmo jeito que gravaria se houvesse a palavra reservada ***mask***.

6.3.6 Semântica da Construção Mapping

Como descrito na seção 6.3.2, a sintaxe da declaração de todo registrador interno exige um identificador de mapeamento do protocolo de acesso ao registrador interno. Logo, se faz necessária a declaração deste mapeamento responsável por indicar quando há o acesso aos registradores internos. Isto é feito através da construção ***mapping***, onde é possível ver sua sintaxe a seguir:

```
mapping <prot_ident> {
     address: <address_ext_reg_ident>;
     data: <data_ext_reg_ident>;
 }
```

Quando o sintetizador se depara com esta construção, ele irá persistir este elemento na memória *mem* associado ao seu identificador. Assim, seja um elemento *mapping* representado através de uma tupla $P = \langle addrReg, dataReg \rangle$, onde *addrReg* é o identificador do registrador externo que contém o valor do endereço do registrador interno que será acessado. Já *dataReg* representa o identificador do registrador externo que contém o dado que será lido ou escrito do registrador interno.

O passo de cômputo que o sintetizador realiza quando encontrar este tipo de construção, com base na sintaxe apresentada acima, é o seguinte:

$$mem\_prot\_sto(prot\_ident, P); \tag{6.17}$$

Através do exemplo a seguir é possível ver com detalhes a semântica da construção em questão:

```
mapping firstRegBank{
    address: regA;
    data: regB;
}
```

A construção acima reflete nos seguintes passos de cômputo:

1. $mem\_prot\_sto(\text{"}firstRegBank\text{"}, (\text{"}regA\text{"}, \text{"}regB\text{"}))$

A partir da próxima seção até a seção 6.3.11 este documento irá descrever a semântica da seção de protocolo da linguagem TDevC.

6.3.7 Semântica da Construção Var

A construção ***var*** (uma abreviação de *Variable*) representa a declaração de variáveis globais que podem ser utilizadas para criação de um contexto interno para a checagem, independente do protocolo de comunicação do dispositivo que foi descrito. Como já detalhado na Seção 6.2 deste documento, as variáveis são utilizadas quando se quer adicionar informações de algum contexto durante validação.

Sua sintaxe é bastante simples e segue o seguinte padrão:

```
var <val_ident> = <int>;
```

A sintaxe desta construção começa com a palavra reservada ***var*** seguida pelo identificador da variável e pelo valor inicial da mesma. Toda variável na TDevC deve ser declarada com um valor inicial. Assim, ao encontrar essa sintaxe o sintetizador irá realizar o seguinte passo de cômputo:

$$mem\_var\_sto(val\_ident, int); \tag{6.18}$$

Por exemplo:

```
var counter = 0;
```

Para este exemplo, o sintetizador realizará o seguinte passo de cômputo:

1. $mem\_var\_sto(\text{"}counter\text{"}, 0)$

6.3.8 Semântica das Construções State e Orthogonal Region

Como visto anteriormente neste capítulo, existem três tipos de estados que podem ser declarados: globalstate, initialstate e state. Eles já foram explicados anteriormente e a semântica da construção será apresentada inferindo-se que o leitor já conhece a fundo essas construções.

Como também já foi detalhado na seção da sintaxe, todos estados contêm pelo menos uma região ortogonal (também apresentada neste capítulo). Por uma questão de definição,

esta seção irá apresentar a semântica com estas duas construções, uma vez que a definição formal de um estado exige a presença de uma região ortogonal.

Assim, temos que a sintaxe da declaração da construção do um estado, para os três tipos, seria:

```
globalstate {
    orthoregion <ortho_ident> {
        <state_type> <state_ident> {...}
        ...
    }
}
```

Nesta sintaxe é importante chamar a atenção para dois detalhes: o primeiro destaca que a primeira declaração de um estado (estado raiz) é obrigatoriamente feita através de um *globalstate* e os demais estados (filhos) são ou *initialstate* ou *state*. Não há um necessidade de checagem de tipos para esses estados, uma vez que sintaticamente não é possível declarar estados filhos como *globalstate* e nem estado raiz como *initialstate* ou *state*. O segundo detalhe diz respeito à declaração da região ortogonal contendo estados filhos. Este é justamente o motivo de a definição da semântica dos estados ser feita em conjunto com as regiões ortogonais. Com isso a definição consegue contemplar a hierarquia da linguagem.

Assim, pode-se definir formalmente o tipo complexo de dados de um estado como sendo uma tupla $S = \langle orthoreg, type \rangle$, onde *orthoreg* é o identificador da região ortogonal à qual o estado faz parte e *type* é o tipo do estado (GLOBALSTATE, INITIALSTATE e STATE). Ao passo que um estado é declarado, a suas regiões ortogonais também são. Assim, também é possível definir o tipo complexo de uma região ortogonal como sendo apenas um identificador do estado ao qual ela compõe.

Dessa maneira, ao encontrar uma construção do tipo *State*, o sintetizador efetua os seguintes passos de cômputo:

1. $mem\_stt\_sto(\text{"}globalstate\text{"}, S)$
2. $mem\_or\_sto(\text{"}ortho\_ident\text{"}, \text{"}globalstate\text{"})$
3. $mem\_stt\_sto(\text{"}state\_ident\text{"}, S')$
4. ...

O passo 1 é o momento onde há a gravação do estado globalstate. Para este passo, $S = \langle \text{" "}, \text{"}GLOBALSTATE\text{"} \rangle$. A referência à região ortogonal se encontra nula ou vazia pelo fato de que um estado raiz (globalstate) não pertence a região ortogonal alguma, como foi definido no capítulo 5.

O passo 2 é o momento onde há o armazenamento da região ortogonal que compõe o estado global. Por esse motivo o identificador *globalstate* foi associado a este armazenamento.

Já o passo 3 é o momento onde ocorre o armazenamento do primeiro estado filho do estado global. Para esse passo, $S' = \langle$"*ortho_ident*", "*state_type*"$\rangle$. É importante destacar que o próximo passo de computo que ocorreria nesta situação seria o armazenamento da região ortogonal do estado filho, mas, por uma questão de simplificação da explicação, este passo não foi detalhado. Isso se seguiria para todos estados filhos do *globalstate*, para os filhos deste e assim por diante.

A construção sintática de declaração de um estado também pode contar com um açúcar sintático para simplificar sua declaração. Nem todo estado pai contém mais de uma região ortogonal. Dessa maneira, caso o projetista queria declarar estados filhos de um estado com apenas uma região ortogonal, o sintetizador automaticamente infere a existência de apenas uma região e efetua o passo de cômputo de armazenamento de uma região ortogonal atribuindo um *hash* gerado automaticamente com o seu identificador, sendo totalmente transparente para o projetista a existência da região.

O exemplo a seguir mostrar como o sintetizador se comporta para os dois casos: com e sem declaração explícita de regiões ortogonais:

```
    globalstate {
2       orthoregion o1 {
            initialstate s1 {
4               initialstate s11{...}
                state s12{...}
6           }
            state s2 {...}
8       }
        orthoregion o2 {
10          initialstate s3 {...}
            state s4 {...}
12      }
    }
```

Este exemplo contém o estado global (linha 1) contendo duas regiões ortogonais (linhas 2 e 9). Na região ortogonal *o1* (linha 2) existem 2 estados: o estado *s1* (linda 3) e o estado *s2* (linha 7). Já o estado *s1*, por sua vez, contém mais dois estados filhos: *s11* e *s12* (linhas 4 e 5, respectivamente). Perceba que no estado *s1* não há declaração de região ortogonal, uma vez que não há mais de uma linha de execução. A região ortogonal *o2*, contém outros estados filhos do estado global: o estados *s3* e o *s4* (linhas 10 e 11, respectivamente).

Para este exemplo, os passos de cômputo realizados pelo sintetizador são:

1. mem_stt_sto("*globalstate*", (" ", "$GLOBALSTATE$"))
2. mem_or_sto("$o1$", "*globalstate*")
3. mem_stt_sto("$s1$", ("$o1$", "$INITIALSTATE$"))
4. mem_or_sto("$134124312HASH$", "$s1$")
5. mem_stt_sto("$s11$", ("$134124312HASH$", "$INITIALSTATE$"))

6. $mem\_stt\_sto(\text{"}s12\text{"}, (\text{"}134124312HASH\text{"}, \text{"}STATE\text{"}))$
7. $mem\_stt\_sto(\text{"}s2\text{"}, (\text{"}o1\text{"}, \text{"}STATE\text{"}))$
8. $mem\_or\_sto(\text{"}o2\text{"}, \text{"}globalstate\text{"})$
9. $mem\_stt\_sto(\text{"}s3\text{"}, (\text{"}o2\text{"}, \text{"}INITIALSTATE\text{"}))$
10. $mem\_stt\_sto(\text{"}s4\text{"}, (\text{"}o2\text{"}, \text{"}STATE\text{"}))$

Para este exemplo foram ignorados todos os passo das construções declaradas dentro dos estados *s11*, *s12*, *s3* e *s4*. Destacam-se neste exemplo os passos 4, 5 e 6 onde o sintetizador identificou a existência de apenas uma linha de execução, criou a região ortogonal, a qual deu o nome de *134124312HASH*, e a associou aos estados s11 e s12.

Uma questão que vale a pena ser dita é em relação ao identificador da região ortogonal gerado automaticamente. Nunca haverá um choque de nomes entre os identificadores gerados e os especificados pelo projetista, uma vez que sintaticamente os identificadores declarados não podem ser iniciados por números e todos os automáticos começam com o número *hash*.

6.3.9 Semântica da Construção ExitPoint

A construção *exitpoint* permite declarar uma transição entre estados irmãos, sendo ela uma construção que compõe a especificação do estado de origem da transição, ou seja, todo *exitpoint* é declarado no estado origem, informando o estado alvo ou destino, como pode ser visto na sintaxe a seguir:

```
...
<state_type> <state_ident> {
    addexitpoint(<target_state_ident>){
        <access_type>(<reg_ident> | any) <oper> <expr>
     }
 }
 ...
```

A declaração de uma transição começa dentro da declaração de um estado, a partir da palavra reservada *addexitpoint* seguida pelo identificador do estado alvo. Dentro de uma declaração de uma transição é declarado também o gatilho que irá disparar a transição. A declaração deste gatilho segue um padrão, começando pelo tipo de acesso ao registrador (READ ou WRITE), seguido pelo identificador do registrador alvo ou da palavra reservada any, informando que se trata de acesso ao dispositivo, independente de qual registrador foi acessado, e tendo como sequencia uma expressão ao qual o valor lido ou escrito será comparado.

Define-se um *exitpoint* como sendo uma 6-tupla $X = \langle OriStt, TgtStt, Acs, Reg, Oper, Exp \rangle$, onde o *OriStt* é o identificador do estado de origem, *TgtStt* é o identificador do estado de destino, *Acs* é o tipo de acesso ao registrador, *Reg* é o identificador do registrador,

Oper é o operador de comparação e *Exp* é a expressão que resultará em um valor que será comparado ao valor lido ou escrito do registrador.

Ao encontrar uma construção deste tipo, o sintetizador efetua os seguintes passos de cômputo:

1. $mem\_extp\_sto(state\_ident - target\_state\_ident - < int >, X)$

onde $X = \langle state\_ident, target\_state\_ident, access\_type, reg\_ident, oper, expr \rangle$ e $< int >$ representa um valor número sequencial.

É possível ver que o identificador da transição (*Exitpoint*) é uma combinação dos identificadores do estado origem com o identificador do estado destino e uma sequência numérica (representada por *0000* no passo 1) para evitar conflitos de nomes. O exemplo a seguir mostrará um exemplo real de como o sintetizador realiza o armazenamento deste tipo de construção sintática:

```
state s1 {
    addexitpoint(s2){
        WRITE(regA) = counter + 1
    }
    addexitpoint(s3){
        READ(regB) = 10
    }
}
```

A construção acima reflete no seguinte passo de cômputo:

1. $mem\_extp\_sto(\text{"}s1 - s2 - 0001\text{"}, (\text{"}s1\text{"}, \text{"}s2\text{"}, \text{"}WRITE\text{"}, \text{"}regA\text{"}, \text{" = "}, \text{"}counter + 1\text{"}))$
2. $mem\_extp\_sto(\text{"}s1 - s3 - 0002\text{"}, (\text{"}s1\text{"}, \text{"}s3\text{"}, \text{"}READ\text{"}, \text{"}regB\text{"}, \text{" = "}, \text{"}10\text{"}))$

6.3.10 Semântica da Construção EntryPoint

Esta construção tem o mesmo objetivo da construção *Exitpoint* uma vez que é um mecanismo que permite a especificação de transições entre estados, entretanto ela é um açúcar sintático. Em algumas situações é interessante que haja uma transição de todos estados de uma região ortogonal para apenas um. Um exemplo disso seria um comando de *reset*. Quando há um *reset* de software o dispositivo volta para o seu estado inicial e, consequentemente, é preciso haver transições para este estado inicial independentemente de onde o ponto da execução esteja.

Assim, diferentemente da construção *Exitpoint*, o *Entrypoint* é declarado dentro na construção de declaração do estado destino e não há necessidade de informar nenhum identificador de outro estado, uma vez que todos os estados da região ortogonal irão fazer parte dessa transição. A sintaxe de um *Entrypoint* pode ser vista a seguir:

```
...
<state_type> <state_ident> {
    addentrypoint {
        <access_type>(<reg_ident>) <oper> <expr>
     }
 }
 ...
```

Muito parecida com a declaração do *exitpoint*, a declaração de um *entrypoint* começa dentro da declaração de um estado, a partir da palavra reservada *addentrypoint* seguida pela declaração do gatilho que irá disparar a transição. A declaração deste gatilho segue o mesmo padrão do *exitpoint*.

Assim, pode-se definir um *entrypoint* como sendo a mesma 6-tupla X definida para o *exitpoint*. A única diferença é que o estado de origem no momento da declaração é vazio ou nulo. Dessa maneira, ao encontrar uma construção deste tipo, o sintetizador efetua os seguintes passos de cômputo:

1. $mem\_entp\_sto(state_{i}dent- < int >, X)$

onde $X = \langle NULL, state\_ident, access\_type, reg\_ident, oper, expr \rangle$ e $< int >$ representa um valor número sequencial.

É importante perceber neste passo de cômputo que o estado da declaração aparece como estado destino, diferentemente da construção *exitpoint*. Isso se dá pelo fato de que o estado que declara um *entrypoint* irá ser sempre o destino das transações de todos os outros estados. E como neste momento ainda não há uma definição dos estados origem e como pode haver mais de um, a referência a um estado origem na definição sempre será vazia ou nula. A seguir será mostrado um exemplo real de como o sintetizador realiza o armazenamento deste tipo de construção sintática:

```
state s1 {
    addentrypoint {
        WRITE(regA) = counter + 10
    }
}
```

A construção acima reflete no seguinte passo de cômputo:

1. $mem\_entp\_sto(\text{"}s1 - 0001\text{"}, (\text{" "}, \text{"}s1\text{"}, \text{"}WRITE\text{"}, \text{"}regA\text{"}, \text{" = "}, \text{"}counter + 10\text{"}))$

Como mencionado anteriormente, o estado *s1* (o estado que declarou o *entrypoint*) aparece na função como destino, e a origem é nula ou vazia, diferentemente de um *exitpoint*. Este tipo de construção serve como entrada para uma futura estruturação e organização da FSM-Monitor nas quais o sintetizador transforma as *entrypoint* em *exitpoint* em uma etapa seguinte à construção dos elementos em memória. Isso será discutido com mais detalhes nas seções que se seguem.

6.3.11 Semântica da Construção Property

Property é uma construção extremamente importante para a linguagem TDevC. A partir dela é possível especificar as propriedades temporais que serão checadas durante a execução do dispositivo. Como as propriedades estão vinculadas aos estados, esta construção também compõe a construção de declaração dos estados. A sintaxe a seguir mostra como é a declaração deste tipo de construção:

```
...
<state_type> <state_ident> {
    addproperty(<type>) <prop_ident> {
        <lang_type>(<formula>)
     }
 }
 ...
```

A declaração de uma propriedade também começa dentro da declaração de um estado, a partir da palavra reservada *addproperty*, seguida pelo tipo da propriedade, que pode ser *WARNING* ou *CRITICAL*. Propriedades do tipo *WARNING* são propriedades que tem como papel informar algo que pode ser prejudicial ou importante para a execução do *driver*, porém sem quebra de protocolo. As propriedades *CRITICAL* são propriedades que violam completamente o protocolo de uso do dispositivo. Assim que uma propriedade deste tipo ocorre, a checagem é abortada uma vez que já não se sabe o estado no qual o dispositivo se encontra.

Dentro da declaração da propriedade é o local da especificação da formula temporal. A sintaxe então requer a declaração do tipo da linguagem de especificação de propriedades a ser usada para especificar a formula, seguida pela fórmula. Atualmente a TDevC suporta apenas formulas LTL.

Define-se uma propriedade através uma 4-tupla $H = \langle state, type, language, formula \rangle$, onde o *state* é o identificador do estado dono da propriedade, *type* é o tipo da propriedade (*WARNING* ou *CRITICAL*), *language* é a linguagem utilizada para expressar a fórmula e *formula* é a expressão textual que representa a fórmula. Portanto, no momento em que o sintetizador encontra esta construção ele efetua o seguinte passo de cômputo:

1. $mem\_prop\_sto(\text{"}prop\_ident\text{"}, H)$

onde $H = (\text{"}state\_ident\text{"}, \text{"}type\text{"}, \text{"}lang\_type\text{"}, \text{"}formula\text{"})$

O exemplo a seguir irá mostrar mais detalho de como esse passo de cômputo ocorre:

```
...
state s1 {
    addproperty(critical) seqWrite {
        ltlf("G (s2 -> (counter < 100))")
     }
 }
 ...
```

onde ocorre o seguinte passo de cômputo:

1. $mem\_prop\_sto(\text{"}seqWrite\text{"}, (\text{"}s1\text{"}, \text{"}CRITICAL\text{"}, \text{"}LTL\text{"}, \text{"}G(s2 \rightarrow (counter < 100))\text{"}))$

Esta propriedade em questão diz que "sempre que o estado *s2* for alcançado, o valor da variável *counter* deve ser menor que 100". Entretanto, esta é uma propriedade fictícia utilizada apenas para exemplificar a declaração de propriedades.

6.3.12 Transformações e Validações

Como visto nas seções anteriores, durante a etapa de *parsing*, o sintetizador realiza o armazenamento dos elementos da linguagem. Entretanto, fica claro que alguns elementos são vinculados a outros elementos apenas por seus identificadores, com base na suposição que os elementos existem e que são os elementos associados são os permitidos.

Assim, fica claro que algumas validações e transformações no modelo especificado são necessárias após a etapa do *parsing*. Estas ações são:

- **Validação de vínculo entre os elementos de interface da TDevC:** valida os identificadores informados nos elementos de interface declarados.
- **Transformação dos Registradores com *format*:** associa os campos declarados nas construções do tipo *format* com os registradores
- **Validação de transições de estados:** valida se as transições respeitam a especificação da FSM-Monitor.
- **Validação de vínculo entre expressões e os elementos da TDevC:** valida se os identificadores utilizados nas expressões são de elementos existentes.
- **Transformação de Entrypoints:** transformações dos *Entrypoints* em *Exitpoints* de todos os estados para o estado alvo em comum.
- **Vinculação direta dos elementos da TDevC:** vínculo direto dos objetos dos elementos associados, substituindo os identificadores.
- **Checagem de não-determinismo:** checa se a máquina de estados gerada é determinística.
- **Checagem de inconsistência entre propriedades:** chega a existência de propriedades contraditórias na minha linha hierárquica.

Cada umas destas validações e transformações serão detalhadas nas seções a seguir. Para apoiar nas validações, definiu-se uma função ***stuck()*** para indicar a interrupção da síntese e que é chamada quando ocorre algum erro de validação da especificação TDevC.

6.3.12.1 Checagem de Vínculo entre os Elementos de Interface da TDevC e Transformação dos Registradores com *Format*

Durante a etapa do *parsing*, registradores externos e internos são armazenados e vinculados a outros elementos como os *format* e *mapping*. Como este vínculo é apenas uma referência ao identificador dos elementos, é preciso checar se os *format* e *mapping* referenciados existem.

O mesmo ocorre para as construções *mapping*. Estas construções fazem referências aos identificadores dos registradores e, consequentemente, deve-se verificar a existência destes elementos referenciados.

Vínculo e Transformação de Registradores com *Format*

Na definição a seguir será utilizado como base a construção de *external register*, porém ela segue o mesma lógica ao ser aplicada à construção *internal register*.

Seja r um identificador, tem-se que $\forall r \in mem\_ereg\_set(), \exists(i,j,k) \mid (i,j,k) = mem\_ereg\_ret(r)$, logo:

1. $j \in mem\_fmt\_set() \rightarrow mem\_ereg\_sto(r,(i,j,f))|f = mem\_fmt\_ret(j)$
2. $(j \neq \text{`` ''} \wedge\ j \notin mem\_fmt\_set()) \rightarrow stuck()$

Para todo r que representa um registrador externo (i,j,k), tem-se que, se j existe no conjunto dos *format* declarados, então os registradores que apontam para este *format* são substituídos pelos novos registradores, incluindo o conjunto de campos declarados no *format* associado (passo 1). Caso não exista um *format* co este identificador, o sintetizador interrompe a síntese (passo 2).

Vínculo de Registradores Internos e *Mapping*

Seja r um identificador, então $\forall r \in mem\_ireg\_set(), \exists(i,j,k,l) \mid (i,j,k,l) = mem\_ireg\_ret(r)$. Dessa maneira:

1. $j \notin mem\_prot\_ret() \rightarrow stuck()$

Para todo registrador interno (i,j,k,l), caso j não se encontre no conjunto dos protocolos de acesso interno declarados através da construção *mapping*, o sintetizador deve interromper a síntese.

Vinculo de *Mapping* e Registradores Externos

Seja m um identificador, então $\forall m \in mem\_prot\_set(), \exists(i,j) \mid (i,j) = mem\_prot\_ret(m)$. Assim:

1. $i \notin mem\_ereg\_set() \rightarrow stuck()$
2. $j \notin mem\_ereg\_set() \rightarrow stuck()$

Se a tupla (i, j) representa um elemento do tipo *mapping*, i e j deve existir no conjuntos dos identificadores dos registradores externos. Caso contrário, o sintetizar deve interromper a síntese (passos 1 e 2).

6.3.12.2 Validação de Transições de Estados

Na Validação de transições de estados são realizadas validações no vínculo com os estados alvos, nas transições para estados da mesma região ortogonal e no vínculo com os registradores informados nos gatilhos das transições.

Vínculo com os Estados Alvos e Checagem de Regiões Ortogonais

Seja t um identificador onde $\forall t \in mem\_extp\_set()$, então:

1. $\exists(os, ts, k, l, o, m) \mid (os, ts, k, l, o, m) = mem\_extp\_ret(t)$
2. $\exists(o1, y1) \mid (o1, y1) = mem\_stt\_ret(os)$

então,

a. $ts \notin mem\_stt\_set() \rightarrow stuck()$
b. $ts \in mem\_stt\_set() \rightarrow \exists(o2, r2) \mid (o2, r2) = mem\_stt\_ret(ts)$
   i. $(o1 \neq o2) \rightarrow stuck$

Em suma, sendo t um *exitpoint* representando a 6-tupla (os, ts, k, l, o, m), existe uma região ortogonal $o1$ na qual o estado origem se encontra. Se o estado destino ts não se encontra no conjunto dos identificadores dos estados, o sintetizador deve interromper a síntese (passo a). Caso o estado destino tenha sido declarado, existe uma região ortogonal $o2$, na qual este estado faz parte (passo b), e caso a região ortogonal $(o1)$ do estado origem não seja igual à região ortogonal $(o2)$ do estado alvo, a síntese deve ser interrompida (passo i).

Vínculo com os Registradores dos Gatilhos das Transições

São validados os gatilhos das transições declaradas tanto através de *exitpoint* quanto de *entrypoint*. Para *exitpoint*, seja t um identificador, onde $\forall t \in mem\_extp\_set()$, $\exists(os, ts, k, l, o, m) \mid (os, ts, k, l, o, m) = mem\_extp\_ret(t)$. Assim:

a. $(l \in mem\_ereg\_set() \wedge l \notin mem\_ireg\_set()) \rightarrow stuck()$

Para *entrypoint*, o processo é semelhante. Seja t um identificador, onde $\forall t \in mem\_entp\_set()$, $\exists(os, ts, k, l, o, m) \mid (os, ts, k, l, o, m) = mem\_entp\_ret(t)$. Assim:

b. $(l \in mem\_ereg\_set() \wedge l \notin mem\_ireg\_set()) \rightarrow stuck()$

Independentemente do tipo de transição de estados, para uma transição t, caso o identificador do registrador l não se encontre nos conjuntos dos identificadores dos registradores internos ou externos, o sintetizador deve interromper a síntese (passos a e b).

6.3.12.3 Validação de Vínculo entre Expressões/Fórmulas e os Elementos da TDevC

Esta checagem verifica se os identificadores utilizados como elementos das expressões foram realmente declarados. Para isso se faz necessária a definição de funções que possibilitam extração de dados em expressões, fórmulas e proposições. São elas:

- $form\_prop : form \rightarrow [prop_0..prop_n]$
- $prop\_exp : prop \rightarrow [exp0..expn]$
- $exp\_elem : exp \rightarrow [elem_0..elem_n]$

A função $form\_prop$ tem como objetivo retornar todas a proposições atômicas de uma formula lógica, como por exemplo:

$$form\_elem(\text{"}(A + 2 == B) \wedge (B + 2 == C)\text{"}) = \{\text{"}A + 2 == B\text{"}, \text{"}B + 2 == C\text{"}\}$$

Já a função *prop_exp* tem como objetivo extrair o conjunto contendo todas as expressões das proposições atômicas: Utilizando uma parte do exemplo anterior, tem-se:

$$form\_exp(\text{"}A + 2 == B\text{"}) = \{\text{"}A + 2\text{"}, \text{"}B\text{"}\}$$

Por fim, a função *exp_elem* tem como objetivo retornar todas as os elementos não numéricos das expressões, isto é, somente os identificadores, como pode ser visto no exemplo a seguir:

$$form\_exp(\text{"}A + 2\text{"}) = \{\text{"}A\text{"}\}$$

Com base nessa função é possível então validar se as expressões estão vinculadas a identificadores válidos.

Vínculo das Expressões de Gatilho nas Transições de Estados

Seja t um identificador, onde $\forall t \in mem\_extp\_set()$, $\exists(os, ts, k, l, m) \mid (os, ts, k, l, m) = mem\_extp\_ret(t)$. Assim:

1. $\forall elem \in exp\_elem(m)$

então,

a. $elem \notin (mem\_ereg\_set() \cup mem\_ireg\_set() \cup mem\_var\_set() \cup mem\_par\_set()) \rightarrow$

$stuck()$

Para toda transição representada através da tupla (os, ts, k, l, m), onde m é expressão gatilho, tem-se que todo elemento *elem* desta expressão deve estar incluído na união dos conjuntos dos identificadores de registradores externo, registradores internos, *var*, e *pattern* (passo a). Caso isso não seja verdade, o sintetizador deve interromper a síntese.

A mesma validação é realizada pra *entrypoint*, a função de acesso ao banco de memória, passando de $mem\_extp\_set()$ para $mem\_entp\_set()$

Vínculo das Fórmulas de Propriedades dos Estados

Seja T um identificador onde $\forall p \in mem\_prop\_set()$, então $\exists(st, ty, f) \mid (st, ty, f) = mem\_prop\_ret(p)$. Assim:

1. $\forall prop \in form\_prop(f), \exists exp \mid exp \in prop\_exp(prop)$
2. $\forall exp,\ \exists elem \mid elem \in exp\_elem(exp)$

então,

a. $elem \notin (mem\_ereg\_set() \cup mem\_ireg\_set() \cup mem\_stt\_set() \cup mem\_var\_set() \cup$

$mem\_par\_set()) \rightarrow stuck()$

Para toda propriedade representada através da tupla (st, ty, f), onde f é a formula lógica que representa a propriedade, tem-se que todo elemento *elem* desta fórmula, extraído através dos passos 1 e 2, deve estar incluído na união dos conjuntos dos identificadores de registradores externo, registradores internos, estados, *var*, e *pattern* (passo a). Caso isso não seja verdade, o sintetizador deve interromper a síntese.

6.3.12.4 Transformação de *Entrypoints*

Já é sabido que os *Entrypoints* também são transições de estados, entretanto elas informam o caminho inverso de uma *Exitpoint*, ou seja, elas são declaradas no estado alvo sem informar o estado origem, uma vez que são todos os outros estados da região ortogonal. Também já é sabido que esta construção é um açúcar sintático. Em outras palavras, a partir dela são gerados inúmeros *exitpoints* automaticamente para representar essas transições de todos os estados para o estado alvo, como pode ser visto na Figura 56 a seguir.

Assim, seja t um identificador, tem-se que $\forall t \in mem\_entp\_set()$:

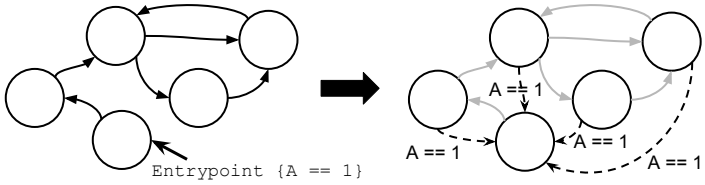


Figura 56 – Transformação de *entrypoints* em *exitpoints*

1. $\exists(null, ts, k, l, m) \mid (null, ts, k, l, m) = mem\_entp\_ret(t)$
2. $\exists(o1, y1) \mid (o1, y1) = mem\_stt\_ret(ts)$

Assim, $\forall s \in mem\_stt\_set(), \exists(o2, y2) \mid (o2, y2) = mem\_stt\_ret(s)$, então:

$$(o1 = o2) \rightarrow mem\_extp\_sto(s\_ts\_0x, (s, ts, k, l, m)) \tag{6.19}$$

Em suma, se t é um identificador de um *entrypoint*, então existe uma 5-tupla representada por este identificador (passo 1) e existe um estado alvo no qual este *entrypoint* foi especificado (passo 2). Destaca-se que não há a necessidade de validar o identificador do estado, uma vez que para que existe o *entrypoit*m, o estado deve existir antes.

Através do estado alvo, conhece-se a região ortogonal do mesmo. Logo, para todos os estados declarados e para todos os que forem da mesma região ortogonal do estado alvo, cria-se um *exitpoint* com todas as informações do *entrypoint* que gerou esta demanda, adicionando apenas o estado destino, como pode ser visto em 6.19.

6.3.12.5 Vinculação Direta dos Elementos da TDevC

Com todos os elementos checados e apontando para as referências corretas, começa então a montagem em memória da máquina de estados. Com a referência já resolvida da etapa anterior da síntese, a FSM-Monitorestá praticamente montada, uma vez que a especificação textual da TDevC é muito semelhante à estrutura da máquina de estados em si. Até as expressões lógicas das transições e as expressões ltl tem suas proposições atômicas apontando para os elementos construídos, apesar de que ainda nesta etapa as expressões ltl são puras concatenações de proposições e operados, e não máquinas de estados. Logo, algumas transformações e checagens simples precisam ser feitas.

Como estes elementos se relacionam através dos seus nomes, estes então passam a referenciar diretamente os elementos construídos e não mais apenas os nomes. A Figura 57 ajuda a explicar esta etapa de montagem do especificação em memória.

Todas os elementos são varridos e todos aqueles que fazer referência a outro elemento através do seu identificar, substituirá esse identificar por uma referência do endereço físico de memória, onde o mesmo poderá ser acessado diretamente como se houve uma cópia do elemento contida dentro de do elemento que faz a referência.

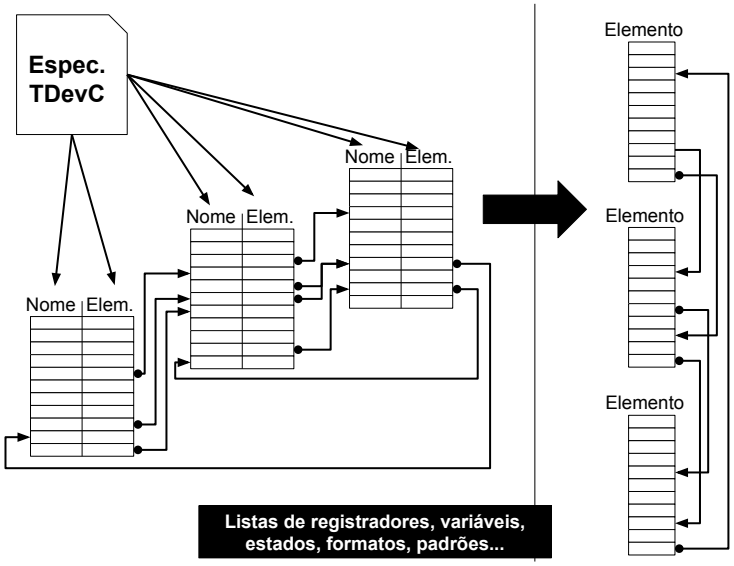


Figura 57 – Montagem de elementos da TDevC em memória

6.3.12.6 Checagem de Não-Determinismo e Inconsistência entre Propriedades

Após estas checagens e transformações, todas as expressões lógicas, sejam transições ou expressões de lógica temporal, são transformadas para um formato intermediário. Porém, para um adequado funcionamento do monitor, é preciso ter certeza de que a FSM-Monitor gerada a partir especificação TDevC é determinística e consistente.

Existe a necessidade de que a FSM-Monitor seja determinística devido à natureza do protocolo que está sendo especificado através da TDevC. Dispositivos operando em uma plataforma têm o seu funcionamento de natureza determinística e, consequentemente, seu protocolo de controle também. Assim, cada comando enviado ao dispositivo deve levá-lo a um estado conhecido. É de suma importância que o *driver* tenha total conhecimento em qual estado o dispositivo estará após acesso a ele.

Já em relação à consistência das propriedades, é fundamental que estas restrições especificadas entre os estados não se contradigam, principalmente as propriedades especificadas entre estados ancestrais ou descendentes. Quando uma propriedade contradiz outra propriedade especificada em um estado ancestral, como o estado pai, por exemplo, nunca haverá uma execução completa do dispositivo (100% de cobertura) sem uma violação de restrição. Isto pode ser proveniente de três situações: ou o modelo foi especificado de maneira errada, ou os requisitos foi entendidos de maneira errada ou, o que é mais crítico, os requisitos foram definidos de maneira errada.

O validação proposta por Carvalho (2016) assume a definição de um monitor inconsistente quando há a possibilidade de ocorrência tanto do não-determinismo quanto da contradição entre propriedades. Assim, para realizar a validação, a abordagem de Carvalho (2016) propõe o fluxo mostrado na Figura 58.

Como já mencionado neste e no capítulo anterior, as transição da FSM-Monitor são

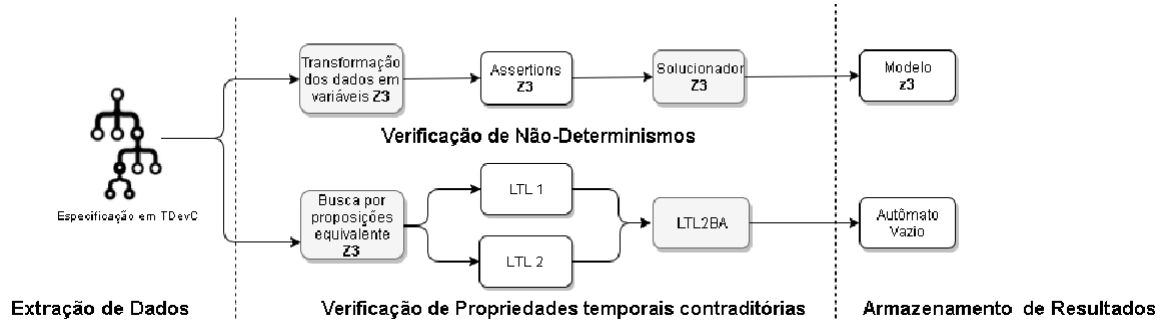


Figura 58 – Fluxo da validação do modelo proposto por Carvalho (2016). Fonte: (CARVALHO, 2016)

realizadas de acordo com a veracidade de expressões lógicas. 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.

Os dados são extraídos de uma especificação em TDevC, são validados com o auxílio do resolvedor de satisfação Z3 (MOURA; BJØRNER, 2008) e o resultado dessa validação é então armazenado. Por questões de simplificação, a partir do formato intermediário da FSM-Monitor, todas as proposições atômicas são substituídas por nomes simplificados e sequenciais no padrão p[x] para as expressões das transições e q[x] para as expressões temporais, onde x é um número inteiro atribuído sequencialmente para cada grupo (p ou q). Ex: A expressão $(A = 3) \wedge (B \neq 2)$ seria traduzida para $(p[1]) \wedge (p[2])$.

Como mencionado, a checagem por não determinismos usa o resolvedor de satisfação *Z3* (MOURA; BJØRNER, 2008). A lista de todos os estados e todas as suas transições no formato simplificado, juntamente com a lista das proposições são enviadas para um *plugin* da *TDevCGen* onde este itera a lista de estados e, para cada um deles, suas transições são comparadas entre si através do *Z3* em busca da possibilidade de equivalência entre elas em um mesmo evento. Se o *Z3* identificar esta equivalência, existe um não determinismo na FSM-Monitor.

Como em uma expressão de transição é possível fazer uso da construção *patterns* e suas máscaras, esta validação leva em consideração a faixa de valores definidas através de máscaras com valores de tipo "*don't care*". Assim, se uma expressão, como por exemplo, "*READ (RegA) = pattrA*"onde *pattrA* é algo do tipo "0000111x", significa que, além de levar em consideração a leitura que será feita no registrador, o validador leva em consideração valor que será lido de *RegA*. Neste caso, ou o valor 14 ou 15.

Desta maneira, o estado que tiver um *exitpoint* com esta expressão como condição, não poderá ter uma outra transição com um expressão como "*READ (RegA) = 14*"ou "*READ (RegA) = 15*", pois seria um não determinismo.

Em resumo, o validador analisa cada transição de cada estado e identifica tanto o tipo de acesso e, se houver a comparação com algum valor, qual a faixa de valores associada.

Já a checagem por inconsistências entre as propriedades temporais também usa o *framework LTL2BA* para validar se as mesmas são contraditórias. Em geral, duas fórmulas ltl f e g são contraditórias se e somente se a união das linguagens de dois autômatos que representam cada ltl for vazia ($L(f) \cup L(g) = \varnothing$), o que indica que não existe sistema que possa satisfazer as propriedades f e g simultaneamente.

Dessa maneira, o validador, em um caminho hierárquico, cria uma propriedade temporária que seja a conjunção das propriedades entre si de um mesmo estado, uma por uma, e das propriedades deste mesmo estado com as propriedades dos seus estados descendentes, também uma por uma (CARVALHO, 2016). Quando a resposta do *framework LTL2BA*, referente a esta propriedade temporária, é equivalente a um autômato *Büchi* vazio ou sem estados finais, isso significa que não existe uma linguagem que seja aceita por este autômato, ou seja, não há uma execução que gere um *trace* sem que viole alguma restrição.

Por fim, quando um não-determinismo ou uma contradição entre propriedades é encontrado, o sintetizador interrompe a síntese e aponta onde houve a inconsistência.

6.4 Resumo

Neste capítulo foi apresentada a linguagem TDevC. Inicialmente foi apresentada a sintaxe da linguagem e na sequencia foi apresentada a semântica da mesma, detalhando como o sintetizador procede quando encontra cada construção da linguagem TDevC. Por fim foi apresentada checagem de tipos e transformações realizadas nos objetos gerados e como os mesmos são vinculados entre si.

7 EXPERIMENTOS E RESULTADOS

Para mostrar a viabilidade e a eficiência da abordagem proposta na detecção de violações de restrições de uso em protocolos de comunicação de alto nível entre *device drivers* e dispositivos, três tipos de experimentos foram realizados.

A primeira série de experimentos foi realizada com base no dispositivo Ethernet DM9000A. Este experimento teve como principal objetivo mostrar a viabilidade e eficácia da técnica. Dessa maneira foi especificado de maneira incremental um modelo em TDevC, foi sintetizado o monitor e foi realizada a validação do *device driver linux* deste dispositivo.

O segundo experimento foi realizado também com base no controlador Ethernet DM9000A e teve como principal objetivo medir o impacto no desempenho do sistema com a utilização da técnica proposta. Para isso foi usada uma ferramenta para medir a largura de banda utilizada pelo dispositivo com e sem o monitor proposto.

Já o terceiro experimento teve como objetivo comparar tanto o desempenho de execução como a produtividade na especificação dos modelos com outra técnica que propõe um mecanismo semelhante (WEISS et al., 2006). Este experimento foi realizado com base em um modelo *SystemC* de um *timer* inserido em uma plataforma *SystemC* especificada no nível TLM.

7.1 Experimentos de Viabilidade e Eficácia

Os experimento de viabilidade e eficácia foi realizado em uma plataforma de hardware sintetizada em FPGA com um processador NiosII executando o sistema operacional μClinux, uma memória, um controlador de LEDs, um controlador Ethernet DM9000A e o monitor MDDC proposto nesta tese. Todos este componentes estão integrados através do barramento Altera Avalon. O diagrama desta plataforma pode ser visto através da Figura 59.

É possível visualizar através da Figura 59 que o MDDC está integrado na plataforma tanto pela porta de observação (BSPI), representada através da linha tracejada, como pela

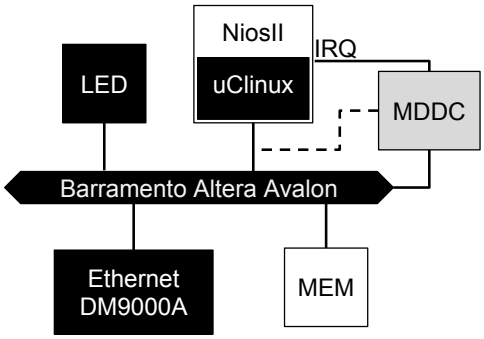


Figura 59 – Infraestrutura real utilizada nos experimentos

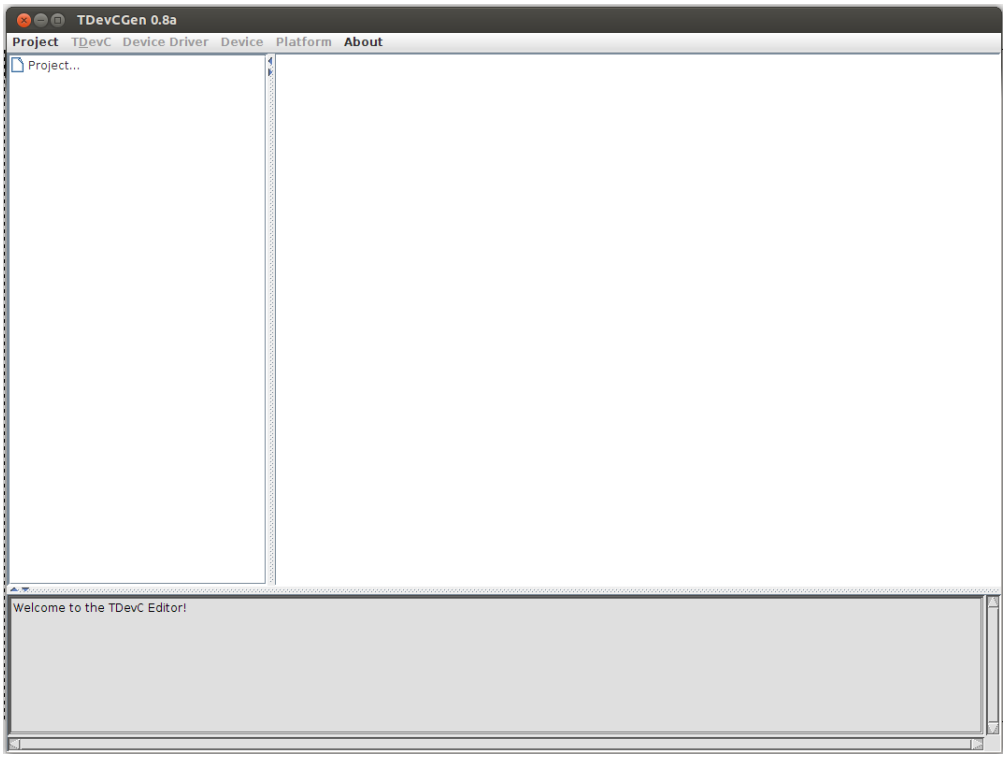


Figura 60 – Tela inicial da ferramenta *TDevCGen*

porta de dispositivo escravo (BSI) conectada ao barramento. Há também uma conexão de interrupção sinalizada pela sigla *IRQ*.

Como mencionado no Capítulo 5, o Ethernet DM9000A é um dispositivo que permite a comunicação entre equipamentos eletrônico e tem como papel em uma plataforma embarcada receber e enviar pacotes de dados de e para dispositivos externos, conectados através de um cabo de par trançado.

Este dispositivo contém uma módulo interno responsável pelo tratamento dos dados recebidos fisicamente através do cabo. Este módulo, chamado de *PHY*, implementa a camada física de rede, com base no modelo OSI.

Assim, este experimento foi realizado neste dispositivo. Para escrita da espcificação em TDevC do protocolo de comunicação do controlador DM9000A, foi utilizada a ferramenta, também proposta neste tesa, *TDevCGen*. Para este primeiro experimento será dada uma pequena introdução sobre a ferramenta.

A Figura 60 mostra a ferramenta *TDevCGen* assim que ela é inicializada. Ela é composta de três seções: a árvore do projeto do lado esquerdo, a área de edição do lado direito e o terminal de status na parta inferior.

Com um projeto aberto, como pode ser visto na Figura 61, na árvore do projeto é possível visualizar todos os componentes da uma especificação em TDevC classificados. É possível ainda visualizar na árvore que este projeto contém duas especificações: *ethernet_dm9000a* e *atuart*.

Na seção de edição é possível visualizar e editar as especificações em TDevC. O ambiente

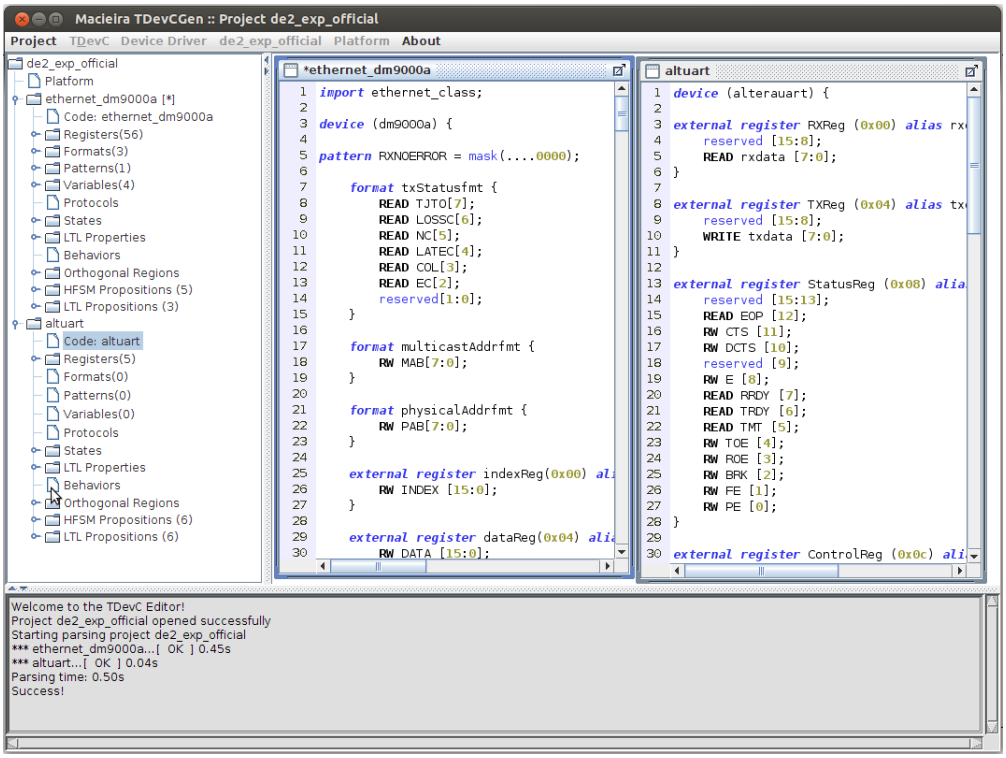


Figura 61 – Tela da ferramenta *TDevCGen* com projeto aberto

de edição conta com destaques na fonte do editor com base na sintaxe da linguagem TDevC. Na Figura 61 as duas especificações estão abertas para edição.

Ainda na Figura 61, no terminal de status é possível visualizar informações sobre o uso da ferramenta e sobre a compilação. Neste exemplo da Figura 61, a terminal de status mostra que o projeto foi aberto com sucesso e que a inicialização e o *parsing* das duas especificações foram realizados com sucesso e em 0,5 segundos.

A ferramente *TDevCGen* ainda permite que sejam visualizadas as máquinas de estados geradas através da síntese das especificações. A Figura 62 mostra duas máquinas de estados da especificação do controlador DM9000A representando a máquina de estados da região ortogonal *phy*(mais acima) e representando o autômato Büchi da restrição *noAccessRST* (mais abaixo). Estas máquinas de estados serão explicadas em detalhes a seguir.

Os experimentos realizados com o controlador Ethernet DM9000A cobriram os serviços de camada física (*PHY*), tais como desligá-la, ligá-la, efetuar *reset*, realizar transmissão e recepção de dados, observar definição do modo de operação e do *link status*. Para atender a este propósito foram especificados no total 56 registradores e 20 estados. Contudo, a especificação em TDevC para este controlador foi realizada em três etapas, de maneira incremental.

7.1.1 Etapa 1

A primeira etapa foi a especificação dos registradores, mapeamento dos registradores internos e do protocolo de comunicação simplificado. A declaração dos registradores e do

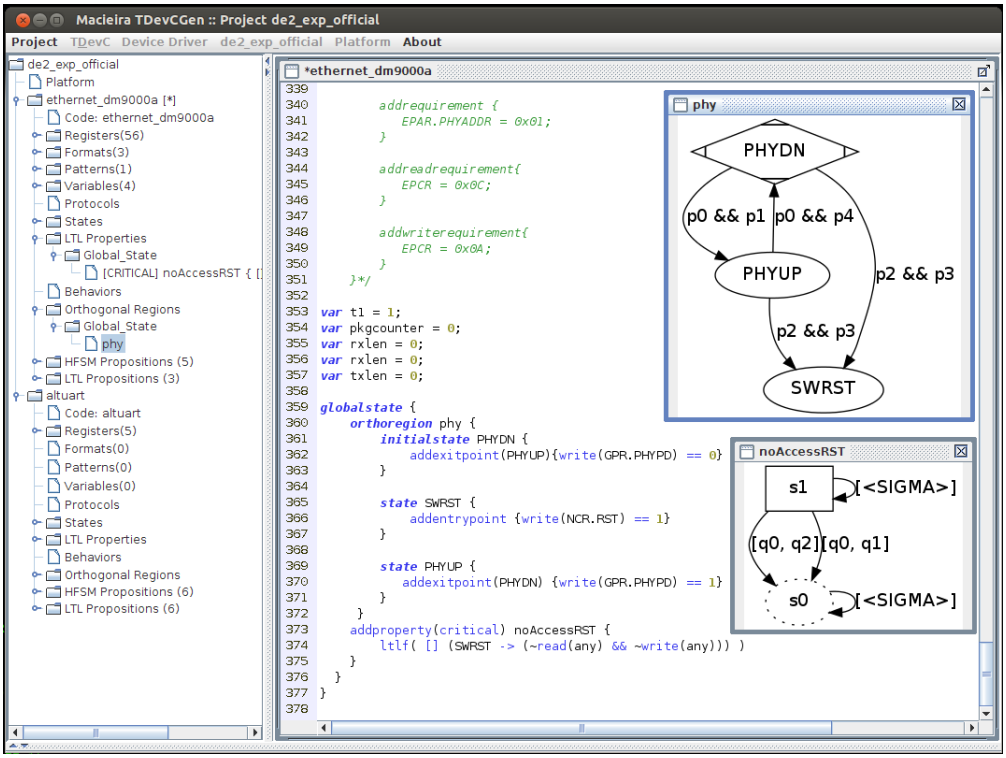


Figura 62 – Tela da ferramenta *TDevCGen* com o detalhes de uma especificação em TDevC

```
1  ...
   globalstate {
3      orthoregion phy {
           initialstate PHYDN {
5              addexitpoint(PHYUP){write(GPR.PHYPD) == 0}
           }
7          state SWRST {
               addentrypoint {write(NCR.RST) == 1}
9          }
           state PHYUP {
11             addexitpoint(PHYDN) {write(GPR.PHYPD) == 1}
           }
13     }
       addproperty(critical) noAccessRST {
15         ltlf( [] (SWRST -> (~read(any) && ~write(any))) )
       }
17 }
     ...
```

Figura 63 – FSM-Monitor especificada para a etapa 1 do primeiro experimento

mapeamentos podem ser vistas no Apêndice B. Por uma questão de organização deste capítulo, serão apresentados aqui apenas as especificações dos protocolos de comunicação, uma vez que a seção de interface da TDevC para este experimento é grande e sua apresentação neste capítulo tornaria a visualização dos experimento mais complexa.

A Figura 63 mostra uma declaração do protocolo de comunicação do controlador do DM9000A. É possível perceber que esta declaração é composta do estado global na Figura 63-linha 2, por uma região ortogonal *phy* na Figura 63-linha 3 e mais três estados: o

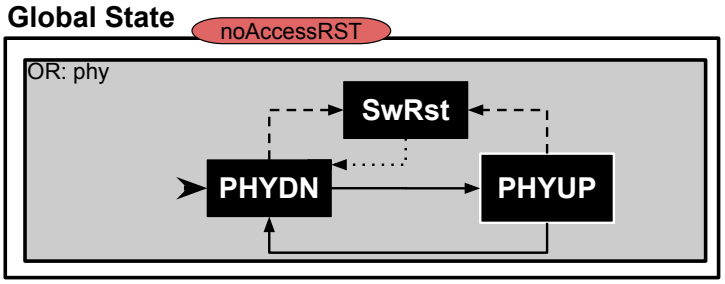


Figura 64 – Representação gráfica da FSM-Monitor especificada para a etapa 1 do primeiro experimento

PHYDN (Figura 63-linha 4), representando o estado do dispositivo em que o módulo *PHY* está desligado, o PHYUP (Figura 63-linha 10), representando o estado do dispositivo em que o módulo *PHY* está ligado, e o SWRST (Figura 63-linha 7), representando o estado do dispositivo após a ocorrência de um *reset* de software.

De acordo com o *datasheet* do controlador DM9000A, quando o dispositivo é inicializado, o estado inicial do módulo *PHY* é o estado de desligado. Assim, como pode ser visto na Figura 63-linha 4, o estado *PHYDN* é o o estado inicial da região ortogonal *phy*.

Para que o módulo *PHY* seja ligado, o *driver* deve escrever no registrador *General Purpose Register* (GPR), mais especificamente no campo *PHYDN* (bit 0), o valor 0 (zero). Esta transição do estado de desligado (PHYDN) para o estado de ligado (PHYUP) foi declarada na especificação e pode ser vista na Figura 63-linha 5.

Já a transição do estado de ligado para o estado de desligado acontece quando o *driver* escreve no registrador *General Purpose Register* (GPR), mais especificamente no campo *PHYDN* (bit 0), o valor 1 (um). Esta transição foi declarada na especificação e pode ser vista na Figura 63-linha 11.

Na Figura 63-linha 8 há declaração de uma transição do tipo *entrypoint*, especificada dentro da declaração do estado *SWRST*. Esta transição ocorre sempre que *driver* escreve no registrador *Network Control Register* (NCR), mais especificamente no campo *RST* (bit 0), o valor 1 (um). Como já mencionando no Capítulo 6, esta transição será convertida em transições com origem em todos os outros estados da região ortogonal *phy* e com destino o estado *SWRST*.

Também é possível ver na especificação mostrada na Figura 63-linha 14 a declaração de uma restrição crítica de uso do protocolo, chamada de *noAccessRST*. De acordo com o manual de uso fornecido pelo fabricante do controlado DM9000A, sempre que o dispositivo entrar em um estado de *reset* não é permitido a realização de escritas ou leituras de dados no dispositivo, até que o mesmo volte ao estado de desligado. Em outros palavras, sempre que o dispositivo estiver no estado de *reset* (*SWRST*), a escrita ou leitura de qualquer valor em seus registradores poderá levar o dispositivo a um estado desconhecido.

Esta restrição foi especificada através da formula LTL "*[] (SWRST -> (~read(any) ~write(any)))*". Esta formula significa que **sempre** (operador temporal "[]") **a presença**

no estado *SWRST* implica (*SWRST ->*) **em não haver um leitura em qualquer registrador** *(~read(any))* **e uma escrita em qualquer registrador** *~write(any).*

A Figura 69 mostra uma representação gráfica da FSM-Monitor especificada na Figura 63. As seta tracejadas representam as transições geradas a partir do *entrypoint* declarado no estado *SWRST*.

É possível perceber tanto na Figura 63 quanto na Figura 69 que não há uma transição de saída do estado *SWRST*. De acordo com o *datasheet* do controlador, existe uma saída automática (sem a atuação do *device driver*) para o estado *PHYDN* 20 μs após o *reset* ser realizado. A técnica proposta suporta parcialmente transições e restrições com base em tempo, uma vez que o monitor já possui módulos desenvolvidos para sinalizar eventos temporais. Entretanto, a sintaxe e semântica da linguagem com suporte à especificação destes eventos ainda não foram totalmente finalizadas.

Para que a abordagem aceite totalmente a especificação de transições e restrições temporais, se faz necessária a especificação tanto da sintaxe como da semântica para que haja o mapeamento destas especificações para o módulo já implementado. Dessa maneira, a transição entre o estado *SWRST* e o *PHUDN* foi representada na Figura 69 através da seta pontinhada.

Após a especificação o MDDC é gerado e integrado manualmente na plataforma. Esta integração, que pode ser vista através do trecho de código em SystemaVerilog mostrado na Figura 65, é realizada de duas maneiras. A primeira é a através da porta de observação (BSPI), onde são criadas replicações de sinais, como uma espécie de bifurcação em uma trilha de sinais. Esta replicação de informações pode ser vista na Figura 65, da linha 1 à linha 8.

A palavra reservada faz a atribuição de um sinal a outro, criando um trilha conectada ao sinal de origem. Por exemplo, a linha 1 da Figura 65 contém a seguinte linha de código: "*assign avalon_bus_address_to_the_MDDC_0 = cpu_0_data_master_address_to_slave;*". Este exemplo resulta em uma atribuição do mesmo valor do sinal *cpu_0_data_master_address_to_sla* ao sinal *avalon_bus_address_to_the_MDDC_0*.

Com esta replicação de sinais, é possível então repassá-los para o MDDC. A Figura 65, da linha 10 à linha 31, mostra a instanciação do MDDC e as suas portas de entrada e saída representando sua interface. A porta de observação (BSPI) é composta pelos sinais especificados entre as linhas 12 e 19 da Figura 65. É possível perceber que os sinais atribuidos a esta porta são exatamente os sinais replicados.

A porta de periférico escravo (BSI) é composta pelos sinais especificados entre as linhas 22 e 30 da Figura 65. Os sinais atribuídos a esta interface não são sinais replicados. Eles são sinais criados a partir do barramento diretamente para o MDDC e vice-versa.

Após a integração, é possível então executar a plataforma com o MDDC incluído. Como mencionado anteriormente, a execução da plataforma conta com um μClinux rodando no processador NiosII. A Figura 66 mostra um *screenshot* do μClinux sendo executado

```
    assign avalon_bus_address_to_the_MDDC_0 =
        cpu_0_data_master_address_to_slave;
2   assign avalon_bus_writedata_to_the_MDDC_0 = cpu_0_data_master_writedata;
    assign avalon_bus_write_to_the_MDDC_0 = cpu_0_data_master_write;
4   assign avalon_bus_readdata_to_the_MDDC_0 = cpu_0_data_master_readdata;
    assign avalon_bus_read_to_the_MDDC_0 = cpu_0_data_master_read;
6   assign avalon_bus_waitrequest_to_the_MDDC_0 = cpu_0_data_master_waitrequest
        ;
    assign avalon_slave_0_wait_counter_eq_0_to_the_MDDC_0 =
        DM9000A_avalon_slave_0_wait_counter_eq_0;
8   assign avalon_slave_0_wait_counter_eq_1_to_the_MDDC_0 =
        DM9000A_avalon_slave_0_wait_counter_eq_1;

10  MDDC_0 the_MDDC_0
    (
12      .avalon_bus_address             (avalon_bus_address_to_the_MDDC_0),
        .avalon_bus_read                (avalon_bus_read_to_the_MDDC_0),
14      .avalon_bus_readdata            (avalon_bus_readdata_to_the_MDDC_0),
        .avalon_bus_waitrequest         (avalon_bus_waitrequest_to_the_MDDC_0),
16      .avalon_bus_write               (avalon_bus_write_to_the_MDDC_0),
        .avalon_bus_writedata           (avalon_bus_writedata_to_the_MDDC_0),
18      .avalon_slave_wait_counter_eq_0 (
            avalon_slave_wait_counter_eq_0_to_the_MDDC_0),
        .avalon_slave_wait_counter_eq_1 (
            avalon_slave_wait_counter_eq_1_to_the_MDDC_0),
20      .clk                            (clk),
        .operMode                       (operMode_from_the_MDDC_0),
22      .chipselect                     (MDDC_0_avalon_slave_0_chipselect),
        .address                        (MDDC_0_avalon_slave_0_address),
24      .interruptLine                  (MDDC_0_avalon_slave_0_irq),
        .pkgCt                          (pkgCt_from_the_MDDC_0),
26      .read                           (MDDC_0_avalon_slave_0_read),
        .readdata                       (MDDC_0_avalon_slave_0_readdata),
28      .rst_n                          (MDDC_0_avalon_slave_0_reset_n),
        .write                          (MDDC_0_avalon_slave_0_write),
30      .writedata                      (MDDC_0_avalon_slave_0_writedata)
    );
```

Figura 65 – Adaptação realizada na plataforma para a integração do Monitor

na plataforma. É possível ver na Figura 66 a inicialização do controlador do dispositivo *DM9000A* a partir da segunda até a sexta linha de *log* do terminal.

Foram realizadas execuções com o MDDC para validar o *device driver* oficial do do dispositivo *DM9000A* fornecido pelo *kernel linux*, versão 2.6. Com a especificação realizada na etapa um deste experimento, nenhuma quebra de restrição foi identificada.

Dessa maneira, por questões de validação da técnica, a restrição foi propositalmente violada. A Figura 67 mostra a função de *reset* do *driver* do controlador do *DM9000A* com a modificação proposital que induz a quebra de restrição de maneira aleatória.

Na Figura 67, das linhas 8 à 15 e a linha 17, estão as modificações realizadas para induzir o erro. Na linhas 8 é atribuído á variável r um valor aleatório de um número inteiro positivo. Assim, na linha 10, é verificado se o valor do *byte* mais significativo deste inteiro é menor do que 50. Se for menor do que 50, o *driver* executa o *reset* do dispositivo com a violação da restrição, ou seja, fazendo um acesso ao dispositivo (linha 13) logo após a solicitação de *reset* (linha 12).

A escolha do valor 50 como sendo limite para entrada na condição de violação da

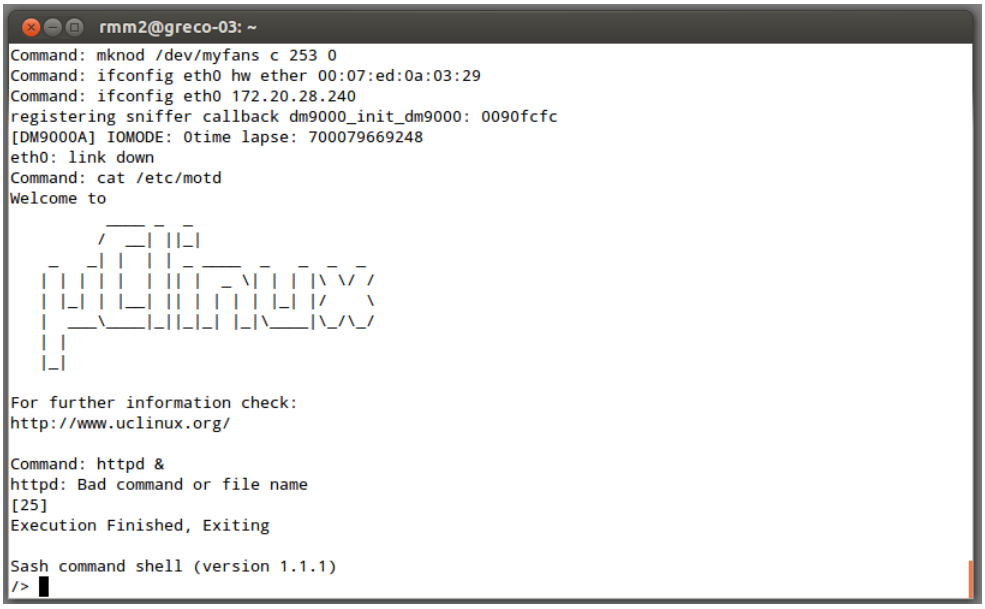


Figura 66 – *Screenshot* do μClinux sendo executado na plataforma

```
1  dm9000_reset(board_info_t * db)
   {
3      int tmp;
       dev_dbg(db->dev, "resetting device\n");
5      /* RESET device */
       writeb(DM9000_NCR, db->io_addr);
7      udelay(200);
       get_random_bytes(&r, sizeof(unsigned int));
9      //******* Erro induzido!
       if ((r >> 24) < 50){
11         printk("\n[DM9000A] Reset Failure Induced!\n");
           writeb(NCR_RST, db->io_data);
13         tmp = ior(db, DM9000_NCR);
       //***********************
15     }else{
           writeb(NCR_RST, db->io_data);
17     }
       udelay(200);
19 }
```

Figura 67 – Trecho da função de *reset* alterada do *device driver* do DM9000A

restrição foi feita também de maneira aleatória. Este valor nada influencia na validação do dispositivo. Ele apenas influencia na chance de que a execução passe por esta condição. Isto apenas acelera ou retarda a chance de que a execução caia neste caso de violação.

Na arquitetura proposta neste experimento, quando há uma violação de restrição, o MDDC envia uma interrupção para o processador NiosII, onde este executa o *callback* da função de interrupção registrada pelo *driver* do MDDC. Esta função pode ser vista na Figura 68.

Quando esta função é invocada, ou seja, quando há uma interrupção devido a detecção de uma violação de restrição por parte do MDDC, a função requisita ao dispositivo MDDC qual foi a violação detectada. O monitor então responde a requisição com um identificador do erro, previamente definida no monitor e catalogada no *driver* do MDDC. A linha 4

```
1  //Driver MDDC IRQ Handler function.
   static irqreturn_t mddcDriver_irqhandler(int irq, void *dev_id/){
3    int failure = 0;
     failure = inl(mddc_ref->error);
5     switch (failure) {
       case MDDC_FAIL_NOACCESSRST
7        printk("\n[MDDC-Driver] Failure Detected: Reset \n");
         break;
9      default:
         printk("\n[MDDC-Driver] Undefined Failure! \n");
11       break;
       }
13   return IRQ_HANDLED;
   }
```

Figura 68 – Função de tratamento de interrupção do *device driver* do Monitor para a etapa 1 do primeiro experimento

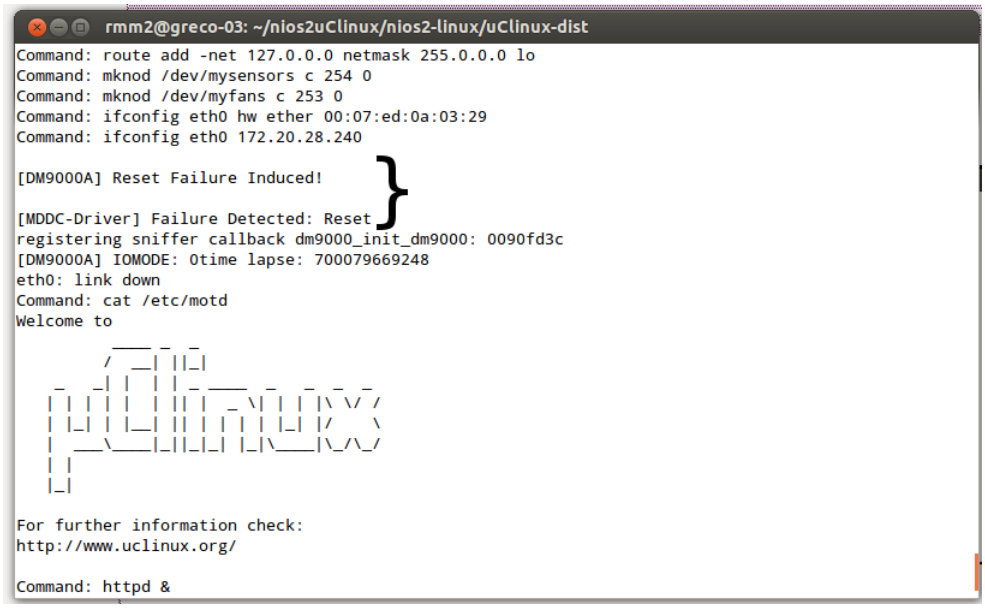


Figura 69 – Terminal do μClinux mostrando a violação e detecção da restrição de *reset*

da Figura 68 mostra o momento em que o *driver* solicita o código da violação. A seguir no código da função de tratamento da interrupção, entre as linhas 5 e 12 da Figura 68, é possível ver o bloco condicional que identifica qual o código da violação e efetua a impressão no terminal do *linux* do texto indicando a violação, como pode ser visto na Figura 67 o texto destacado pela chave.

É importante chamar a atenção para o fato de que são dois componentes isolados que efetuam a impressão dos textos n o terminal. Isso mostra que a ação de impressão do texto realizada pelo *driver* do MDDC disparada puramente devido a detecção da violação pela porta de observação. A impressão do texto foi apenas uma escolha de reação para que fosse possível visualizar e reportar nesta tese a detecção.

Em testes isolados, sem a necessidade de reportar textualmente a ocorrência da violação, esta abordagem também identificou a violação através de LEDs instalados na placa de prototipação onde a plataforma está sendo executada. Assim, não houve qualquer

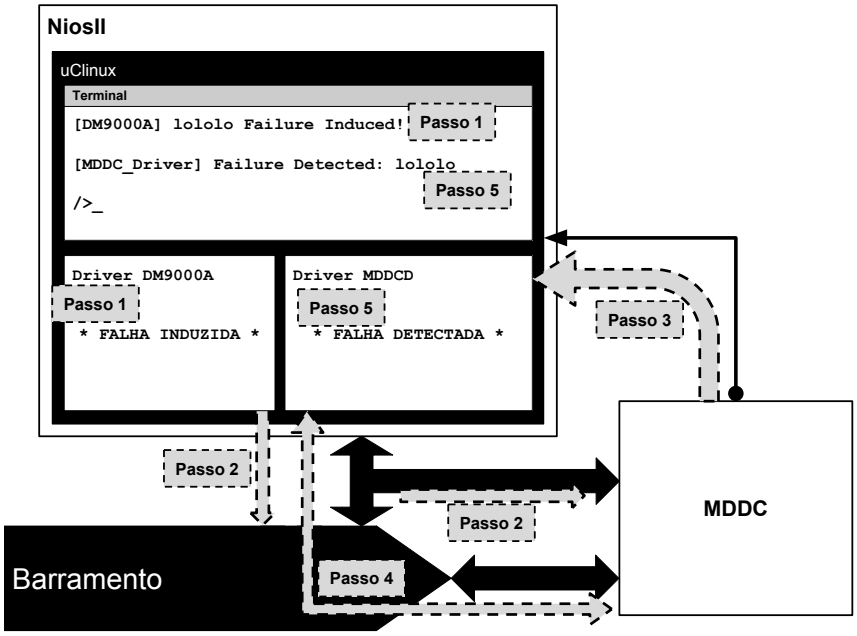


Figura 70 – Sequencia de passos do funcionamento da indução e captura de violações de restrições

interferência de software relativa à checagem do monitor, incluindo a interrupção do MDDC no processador NiosII.

A Figura 70 mostra a dinâmica da utilização do *driver* do MDDC para imprimir no terminal a mensagem de identificação da violação. A Figura 70 mostra uma arquitetura composta pelo processador NiosII e o MDDC conectados em um barramento. No passo 1 sinalizado na Figura 70 é possível ver que é o exato momento em que o erro induzido ocorre no *driver* do controlador do *DM9000A*. Neste mesmo instante há a impressão do texto "*[DM9000A] Reset Failure Induced!*"no terminal do *Linux*. Logo em seguida, no passo 2 da Figura 70, o *driver* do controlador do *DM9000A* envia a sequencia de comandos indesejados para o dispositivo DM9000A. Automaticamente o MDDC captura esta sequencia de comandos. No passo 3 da Figura 70, o MDDC então identifica a violação e dispara uma interrupção para o processador NiosII. O processador então repassa essa interrupção para o *driver* do MDDC. No passo 4 da Figura 70, o *driver* do monitor trata a interrupção requisitando ao MDDC o identificador do erro via porta escrava do monitor. Por fim, no passo 5 da Figura 70, o *driver* do MDDC, já com a identificação da violação, imprime no terminal do *Linux* exatamente qual foi esta violação detectada.

Para esta primeira etapa do experimento foi induzida uma violação da restrição *noAccessRST* do estado global. A função de *reset* do *driver* do controlador *DM9000A* é invocada no inicializador do μClinux. Dessa maneira o *Linux* foi inicializado 50 vezes. Dentre estas 50 vezes, o erro ocorreu 23 vezes e a esta violação foi detectada todas as 23 vezes, no momento em que ela ocorreu. É importante destacar que não houveram falsos-positivos, o que significa que o MDDC somente sinalizou a violação quando ela

```
globalstate {
...
//*****************************[LINK MODE]*******************************
    orthoregion linkstatus {
        initialstate DOWN {
            addexitpoint(UP){read(NSR.LINKST) == 1}
        }
        state UP {
            addexitpoint(DOWN){read(NSR.LINKST) == 0}
        }
    }
    //*************************[OPER MODE]*****************************
    orthoregion opermode {
        initialstate UNDEF_OPER_MODE {
            addexitpoint(OPER16BITS){read(ISR.IOMODE) == 0}
            addexitpoint(OPER8BITS){read(ISR.IOMODE) == 1}
            addentrypoint{write(NCR.RST) == 1}
        }
        state OPER16BITS {
            addexitpoint(OPER8BITS){read(ISR.IOMODE) == 1}
        }
        state OPER8BITS {
            addexitpoint(OPER16BITS){read(ISR.IOMODE) == 0}
        }
    }
    //*****************************[PHY]*********************************
    orthoregion phy {
        initialstate PHYDN {
            addexitpoint(PHYUP){write(GPR.PHYPD) == 0}
        }
        state SWRST {
            addentrypoint {write(NCR.RST) == 1}
        }
        state PHYUP {
            addexitpoint(PHYDN) {write(GPR.PHYPD) == 1}
            addproperty(critical) UndefinedOperMode {
                ltlf([](~UNDEF_OPER_MODE))
            }
        }
    }
    addproperty(critical) noAccessRST {
        ltlf( [] (SWRST -> (~read(any) && ~write(any))) )
    }
  }
...
```

Figura 71 – FSM-Monitor extraída da especificação em TDevC utilizada na etapa 2 do primeiro experimento

realmente aconteceu.

7.1.2 Etapa 2

Na segunda etapa deste experimento a especificação FSM-Monitor foi incrementada. A Figura 71 mostra a declaração de mais duas regiões ortogonais: a *linkstatus*, da linha 4 à linha 11, e a *opermode*, da linha 13 a 25.

A região ortogonal *linkstatus*, especifica quando um link (conexão) está ativo no momento. Esta região ortogonal foi especificada apenas para demostrar que é possível incrementar a especificação sem alterar ou modificar os elementos que já foi especificados anteriormente. Após a especificação da região ortogonal *linkstatus*, o protocolo referente

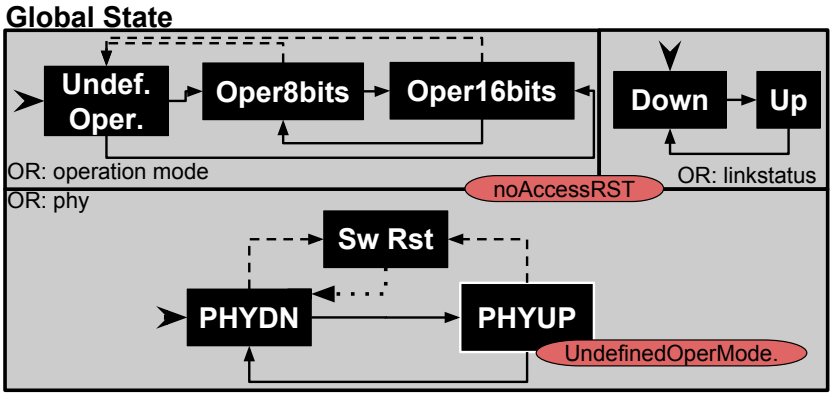


Figura 72 – Representação gráfica da FSM-Monitor extraída da especificação em TDevC utilizada na etapa 2 do primeiro experimento

ao estado de operação do controlador *DM9000A* continua inalterado e funcionando do mesmo modo que foi apresentado na etapa anterior.

já a região ortogonal *opermode* descreve a escolha do modo de operação entre 8 e 16 bits. O dispositivo *Ethernet DM9000A* suporta uma conexão ao barramento da plataforma tanto através de uma porta de 8 bits quanto uma de 16 bits. Assim, quando o controlador é inicializado, o *driver* requisita tal informação ao dispositivo.

Enquanto o *driver* do controlador *DM9000A* não efetuar esta leitura, para ele o modo de operação permanece indefinido, exatamente como o estado inicial *UNDEF_OPER_MODE* da região ortogonal *opermode*. É o modo de operação que irá ditar como o envio e o recebimento de dados são realizados.

Por este fato, foi especificada uma restrição de uso do protocolo no estado *PHYUP*, da região ortogonal *phy*. Esta restrição pode ser vista da linha 36 à linha 38 da Figura 71. É importante reforçar que esta restrição somente será checada quando a execução do dispositivo estiver no estado *PHYUP*. Fora deste estado esta restrição é irrelevante.

A Figura 72 apresenta uma representação gráfica da FSM-Monitor especificada em TDevC mostrada na Figura 71. Percebe-se que agora existem duas restrições, sendo a *UndefinedOperMode* válida unicamente no estado *PHYUP* e a *noAccessRST* válida em todos os estados, uma vez ela foi declarada pelo estado global.

Como o *driver linux* não contém violações em relação ao modo de operação, novamente a restrição precisou ser induzida. A Figura 73 mostra o *driver* do controlador *DM9000A* implementado para *kernel Linux*. Como mencionado anteriormente, assim que o dispositivo *Ethernet* é ligado, o módulo *PHY* se encontra no estado *PHUDN*. Por este motivo o seu *driver* precisa ligá-lo para que o dispositivo possa transmitir e receber pacotes de rede. A Figura 73, da linha 15 à 17, mostra os acessos realizados pelo *driver* do controlador do *DM9000A*, escrevendo o valor 0 (zero) no registrador GPR.

Antes de ligar o módulo *PHY* do controlador da Ethernet, o *driver* requisita do dispositivo qual o modo de operação no qual o controlador foi integrado à plataforma.

```
1  static void dm9000_init_dm9000(struct net_device *dev)
   {
3      unsigned long ini, end;
       ini = jiffies;
5      end = jiffies;
       board_info_t *db = netdev_priv(dev);
7      unsigned int imr;
       dm9000_dbg(db, 1, "entering %s\n", __func__);
9      /* I/O mode */
       //******* Erro induzido!
11     //db->io_mode = ior(db, DM9000_ISR) >> 6; /* ISR bit7:6 keeps I/O mode */
       printk("[DM9000A] Undef OperMode Failure Induced!");
13     //***************************
       /* GPIO0 on pre-activate PHY */
15     iow(db, DM9000_GPR, 0); /* REG_1F bit0 activate phyxcer */
       iow(db, DM9000_GPCR, GPCR_GEP_CNTL);    /* Let GPIO0 output */
17     iow(db, DM9000_GPR, 0); /* Enable PHY */
...
```

Figura 73 – Trecho da função de inicialização alterada do *device driver* do DM9000A

Como pode ser visto na Figura 73-linha 11, o *driver* faz uma leitura no registrador *Interrupt Status Register* (ISR), mais especificamente no bit 7 (através da operação »> 6"). Este bit indica se o controlador se encontra em modo 16-bits (quando o valor lido é zero) ou 8-bits (quando o valor lido é um). Para induzir a violação, este comando foi comentado no código do *device driver* do controlador.

Antes de ler o conteúdo do registrador *ISR*, o *driver* não tem ideia de qual modo de operação o dispositivo se encontra, e isto é fundamental para que o *driver* configure suas funções de leitura e escrita. Por isso, é essencial que o modo de operação seja conhecido pelo *device driver* do controlador *DM9000A* quando a *PHY* estiver ligada. É exatamente isto que formula LTL da restrição diz: "[](~UNDEF_OPER_MODE)", ou seja, **sempre** (operador []) **o estado *UNDEF_OPER_MODE* não deve ser o estado corrente** (~UNDEF_OPER_MODE)). É importante lembrar que, como esta restrição foi declarada no estado *PHYUP*, o operador temporal **sempre** se restringe aos momentos em que o estado *PHYUP* é corrente. A hierarquia dá este poder de composição de propriedades de uma maneira mais fluida e mais clara.

Antes de executar a plataforma com o MDDC com a FSM-Monitor incrementada, a função de tratamento de interrupção do monitor foi atualizada, como pode ser visto na Figura 74, da linha 5 à linha 7. Foi apenas adicionado o identificado da violação para a restrição de modo de operação indefinida.

Após o incremento da FSM-Monitor a plataforma foi executada e a violações foi capturada. Com a modificação do *driver* do *DM9000A*, durante a inicialização do dispositivo, há a ativação do módulo *PHY*, porém sem antes haver a leitura do modo de operação. Isto implica em uma transição para o estado *PHYUP* sem que seja conhecido o modo de operação da controlador, tornando a violação aparente para o monitor. É possível visualizar o momento na detecção através do terminal *Linux* mostrado na Figura 75.

Para validar a hierarquia e declaração de restrições específicas a estados de maneira

```
    switch (failure) {
2    case MDDC_FAIL_NOACCESSRST
       printk("\n[MDDC-Driver] Failure Detected: Reset \n");
4      break;
     case MDDC_FAIL_UNDEFINEDOPERMODE
6      printk("\n[MDDC-Driver] Failure Detected: Undef OperMode");
       break;
8    default:
       printk("\n[MDDC-Driver] Undefined Failure! \n");
10     break;
     }
12  return IRQ_HANDLED;
}
```

Figura 74 – Função de tratamento de interrupção do *device driver* do Monitor para a etapa 2 do primeiro experimento

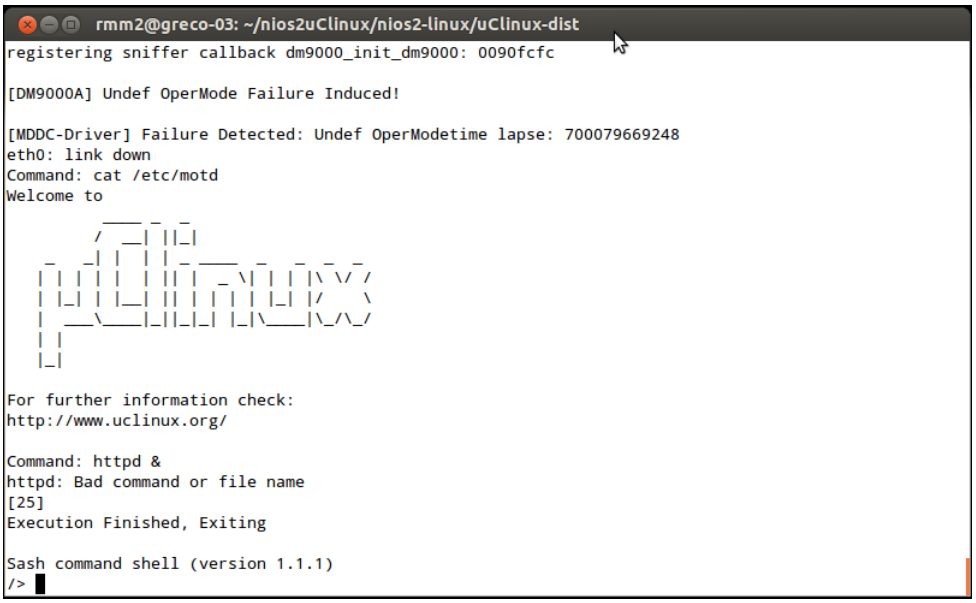


Figura 75 – Terminal do μClinux mostrando a violação e detecção da restrição de modo de operação indefinido

mais clara, foi realizada uma alteração no *driver* do controlador *DM9000A* para que não mais o módulo *PHY* fosse ligado na inicialização do *device driver*. Dessa maneira, todo trecho de código mostrado na Figura 73 entre as linhas 15 e 17 foi comentado e o mesmo trecho foi copiado para a função de desativação do módulo *PHY*, logo acima da escrita do valor 1 (um) no registrador *GPR*, comando este utilizado para desligar o módulo *PHY*, como pode ser visto na Figura 76.

Dessa maneira, sempre que o comando nativo do *Linux ifconfig eth0 down* for chamado, *driver* do *DM9000A* irá ligar e imediatamente na sequencia irá desligar o módulo *PHY*. Em contrapartida, durante a inicialização do controlador, o seu *driver* deveria fazer apenas aa leitura do modo de operação, se a violação não fosse induzida.

Essa alteração permite que a violação seja induzida, mas, como o dispositivo não se encontra no estado *PHYUP*, o monitor não irá detectá-la até que o *DM9000A* seja ligado. De fato, até que ele seja ligado, a violação induzida não representa uma violação até o

```
1         board_info_t *db = netdev_priv(dev);
          /* RESET device */
3
          iow(db, DM9000_GPR, 0); /* REG_1F bit0 activate phyxcer */
5         iow(db, DM9000_GPCR, GPCR_GEP_CNTL);    /* Let GPIO0 output */
          iow(db, DM9000_GPR, 0); /* Enable PHY */
7
          dm9000_phy_write(dev, 0, MII_BMCR, BMCR_RESET); /* PHY RESET */
9         iow(db, DM9000_GPR, 0x01);      /* Power-Down PHY */
          iow(db, DM9000_IMR, IMR_PAR);   /* Disable all interrupt */
11        iow(db, DM9000_RCR, 0x00);      /* Disable RX */
```

Figura 76 – Trecho da função de desligamento do módulo *PHY* alterada do *device driver* do DM9000A

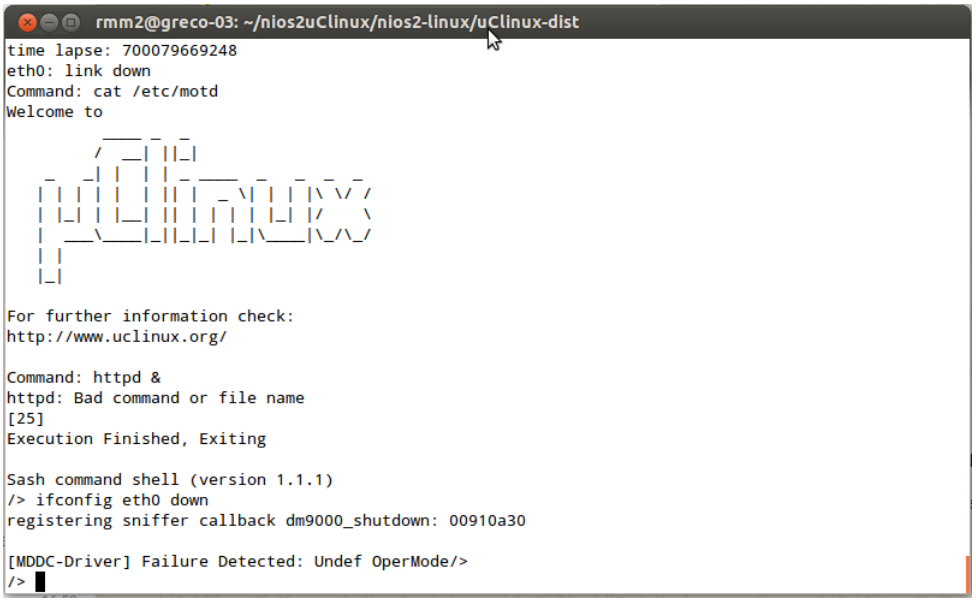


Figura 77 – Terminal do μClinux mostrando a violação e detecção da restrição de modo de operação indefinido com o desligamento da interface

momento, uma vez que em qualquer outro estado que não seja o *PHYUP*, não interessa o conhecimento do modo de operação do dispositivo *Ethernet*.

A Figura 77 mostra que quando a função de inicialização do *driver* é executada, a violação é induzida. Porém, é possível perceber também através da Figura 77 que somente quando o comando *ifconfig eth0 down*, que, devido à modificação do *driver* do *DM9000A*, liga o módulo *PHY* (transição para o estado *PHYUP*) antes de desligá-lo outra vez, a violação passa a existir e o MDDC detectá-la. Isto mostra a viabilidade do uso de restrições dentro dos estados.

7.1.3 Etapa 3

A etapa 3 da especificação em TDevC do controlador *DM9000A* se concentrou no refinamento do estado *PHYUP*. Este refinamento pode ser visto na especificação do estado *PHYUP* através das Figuras 79, 81 e a 82.

```
1     ...
      var t1 = 1;
3     var pkgcounter = 0;
      var rxlen = 0;
5     var rxlowlen = 0;
      var txlen = 0;
7     var txpkgcounter = 0;

9     globalstate { ... }
```

Figura 78 – Declaração de variáveis da especificação em TDevC utilizada na etapa 3 do primeiro experimento

Para suportar a especificação da transmissão de pacotes de rede, foram especificadas algumas variáveis de contexto da validação. Essas variáveis, que podem ser vistas na Figura 78, permitem que o monitor tenha um contexto quanto ao estado da validação, não apenas ao contexto do estado do dispositivos. Por exemplo, com essas variáveis é possível realizar a contagem de pacotes transmitidos. Esta quantidade é indiferente para o estado ou modo de operação do dispositivo *DM9000A*, porém, para a validação, essa informação é fundamental. Cada variável será explicada a medida em que as regiões ortogonais do estado *PHYUP* forem detalhadas.

O estado *PHYUP* foi explodido em 3 regiões ortogonais: *transmissionConfig*, responsável pela escrita da carga útil e configuração do pacote de rede a ser transmitido, *transmissionExec*, responsável pelo envio físico do pacote de rede, e *receive*, responsável pelo recebimento físico de pacote de rede.

A separação entre a configuração (montagem) e o envio físico do pacote, que pode ser vista na Figura 79-linhas 3 e 29, se dá pelo fato de que o controlador *DM9000A* permite a configuração e o envio de dois pacotes de forma simultânea. Isso quer dizer que enquanto um pacote está sendo enviado para outro dispositivo *Ethernet*, a CPU pode escrever na memória do *DM9000A* o novo pacote que será transmitido em seguida.

Ainda observando a Figura 79 é possível ver que, nas linhas 32, 33 e 39, algumas transições de estados foram comentadas. Essas transições foram escritas para validar a checagem por não-determinismos realizada na antes da geração do monitor. De maneira proposital, estas transições forem inseridas para verificar se a técnica iria identificar o não determinismo. Todos os não-determinismos foram detectados como pode ser vistos na Figura 80. A Figura 80 é saída do validador e através dela pode-se ver também os contra-exemplos, mostrando exatamente o porquê da existência do não-determinismo.

A primeira linha da tabela mostrada na Figura 80, diz respeito ao não-determinismos encontrado no estado *WAITREADY*, da região ortogonal *transmissionExec*. Como pode ser visto na Figura 79-linha 32, há uma comparação do valor escrito no registrador *CHIPR* com o pattern *RXNOERROR*, onde sua declaração é "*pattern RXNOERROR = mask(xxxx0000);*".

Exatamente como mostra o contra-exemplo da Figura 80, a possibilidade do bit 5

```
state PHYUP {
    addexitpoint(PHYDN) {write(GPR.PHYPD) == 1}
    orthoregion transmissionConfig {
        initialstate WRITEDATA {
            addexitpoint(SETLENGTH) {
                write(MWCMDR)
                assign{txpkgcounter = txpkgcounter + 1}
            }
        }
        state SETLENGTH {
            addexitpoint(SETREADY) {
                write(MDRAHR) || write(MDRALR)
            }
            addexitpoint(SETLENGTH) {
                write(MWCMDR)
                assign{txpkgcounter = txpkgcounter + 1}
            }
        }
        state SETREADY {
            addexitpoint(SETREADY) {
                write(MDRAHR) || write(MDRALR)
            }
            addexitpoint(WRITEDATA) {
                write(TCR.TXREQ) == 1
                assign{txlen = (MDRAHR << 8) + MDRALR}
            }
        }
    }
    orthoregion transmissionExec {
        initialstate WAITREADY {
            addexitpoint(SENDDATA){write(TCR.TXREQ) == 1}
            //addexitpoint(SENDDATA){write(CHIPR) == RXNOERROR} TESTE PARA
                VALIDACAO
            //addexitpoint(SENDDATA){write(CHIPR) == 32} TESTE PARA VALIDACAO
        }
        state SENDDATA {
            addexitpoint(WAITREADY) {
                    read(NSR.TX1END) == 1 || read(NSR.TX2END) == 1
                }
            //addexitpoint(WAITREADY){read(NSR) == t1} TESTE PARA VALIDACAO
        }
    }
```

Figura 79 – Declaração parcial da FSM-Monitor da especificação em TDevC utilizada na etapa 3 do primeiro experimento - Parte 1

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
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]

Figura 80 – Resultado da checagem por não determinismos na especificação em TDevC utilizada na etapa 3 do primeiro experimento

do padrão *RXNOERROR* ter o valor 1 (um) e todos os outros terem o valor 0 (zero) coincidiria com o valor de 32 atribuido ao mesmo registrado na outra transição, mostrada na Figura 79-linha 33.

Já a segunda linha da tabela mostrada na Figura 80, diz respeito ao não-determinismo encontrado no estado *SENDDATA*, também da região ortogonal *transmissionExec*. Neste caso, como pode ser visto na Figura 79-linha 39, há uma checagem de leitura de uma valor no registrador *NSR*. Este valor esta sendo comparado a uma variável de contexto, que pode assumir qualquer valor durante a execução do monitor. Do mesmo, na Figura 79-linha 37, há outra transição disparada com base na comparação do valor lido dos dois campos *TX1END* (bit 2) e o *TX2END* (bit 3) do mesmo registrador *NSR*.

Como mostra o contra-exemplo na Figura 80, basta apenas o campo *TX2END* ter o valor 1 (um) e a variável ter o seu bit 3 com este mesmo valor para que haja o não-determinismo. É importante ressaltar que este é apenas um contra-exemplo. Na verdade poderia, por exemplo também, o campo *TX1END* com o valor 0 (zero) e a variável ter o valor 8 (binário: 00001000), coincidindo assim os valores do bit 2.

Algumas restrição também foram inseridas no estado *PHYUP*, mas três merecem destaque e podem ser vistas na especificação da Figura 82. Uma delas descreve uma restrição alertada pelo *datasheet* referente à configuração e à transmissão de pacotes.

A restrição, chamada de *NeverLenAndSend*, diz que enquanto um pacote está sendo transmitido (Figura 79, estado *SendData* da região ortogonal *transmissionExec*), a configuração de um novo dado não pode modificar o valor no registrador referente ao tamanho do pacote (Figura 79, estado *SetLength* da região ortogonal *transmissionConfig*). Ou seja, mesmo depois de ter escrito o novo dado na memória da *Ethernet*, o *driver*, deve aguardar que o pacote anterior seja transmitido para continuar a configuração de um novo pacote na memória do dispositivo, sinalizando-o como pronto para envio (Figura 79, estado *SetReady* da região ortogonal *transmissionConfig*).

Outra restrição diz respeito ao tamanho do pacote escrito pela CPU e o tamanho informado. A CPU sempre deve especificar corretamente o tamanho do pacote igual à quantidade de bytes escritos. Assim, durante a escrita do pacote na memória local do controlador *Ethernet* o monitor conta a quantidade de bytes escritos e verifica se o valor informado na configuração é igual ao que realmente foi escrito. Esta é a restrição *TXLenPPT*, que pode ser vista na Figura 82.

A contagem dos *bytes* escritos na memória do controlador pode ser vista na Figura 79 nas linhas 7 e 16. As atribuições sempre ocorrem em transições de estados. Estas transições especificamente são disparadas com a escrita de um dado na memória do dispostivo *Ethernet*, através das escrita no registrador *Memory Data Write Command with Address Increment Register* (MWCMDR).

O registrador *MWCMDR* permite a escrita de um dado na memória do *DM9000A* e automaticamente é feito o incremento no endereço corrente nesta memória, sem que

```
state PHYUP {
    ...
        orthoregion receive {
        //Reception
            initialstate WAITNEWDATA {//RXPKGIDLE  {
                addexitpoint(READSTATUS) {
                    read(MRCMD) == 0x01
                }
                addexitpoint(WAITNEWDATA) {
                    read(MRCMD) == 0x00
                }
                addexitpoint(RXPKGHEADERERR) {
                    read(MRCMDX) != 0x00 && read(MRCMDX) != 0x01
                }
            }
            state READSTATUS{//RXPKGSTATUS {
                addexitpoint(READLOWLENGTH) {
                    read(MRCMD) == RXNOERROR
                }
            }
            state READLOWLENGTH {//RXPKGLOWLEN {
                addexitpoint(READHIGHLENGTH) {
                    read(MRCMD)
                    assign{rxlowlen = MRCMD}
                }
            }
            state READHIGHLENGTH { // RXPKGHIGHLEN {
                addexitpoint(READPAYLOAD) {
                    read(MRCMD)
                    assign{rxlen = (MRCMD << 8) + rxlowlen
                    assign{pkgcounter = 0}
                }
            }
            state READPAYLOAD {//RXPKGPAYLOAD {
                addexitpoint(READPAYLOAD) {
                    read(MRCMD) && pkgcounter < rxlen
                    assign{pkgcounter = pkgcounter + 1}
                }
                addexitpoint(READPAYLOAD) {
                    read(MRCMDX) && pkgcounter < rxlen
                }

                addexitpoint(READSTATUS) {
                    (read(MRCMDX) == 0x01 || read(MRCMD) == 0x01) && pkgcounter
                        == rxlen
                }

                addexitpoint(WAITNEWDATA) {
                    (read(MRCMDX) == 0x00 || read(MRCMD) == 0x00) && pkgcounter
                        == rxlen
                }

                addexitpoint(RXPKGHEADERERR) {
                    (read(MRCMD) != 0x00 && read(MRCMDX) != 0x01) && pkgcounter
                        == rxlen
                }

            }
            state RXPKGHEADERERR {
                addexitpoint(RXPKGHEADERERR) {
                    read(MRCMDX) || read(MRCMD)
                }
                addproperty(critical) readInError {
                    ltlf( []~(read(MRCMD)) )
                }
            }
        }
...
```

Figura 81 – Declaração parcial da FSM-Monitor da especificação em TDevC utilizada na etapa 3 do primeiro experimento - Parte 2

```
state PHYUP {
    ...
    addproperty(critical) UndefinedOperMode {
        ltlf([](~UNDEF_OPER_MODE))
    }
    addproperty(critical) WriteBeforeLen {
        ltlf(
            [](write(MDRALR) || write(MDRAHR)) -> ~WRITEDATA)
            )
        }
    addproperty(critical) NeverLenAndSend {
        ltlf(
            []( ~(SENDDATA && SETREADY) )
        )
    }
    addproperty(critical) TXLenPPT {
       ltlf(
            [](SENDDATA -> ((OPER16BITS -> ((txlen == txpkgcounter) || (txlen
                == txpkgcounter-1))) && (OPER8BITS -> (txlen == txpkgcounter)))
                )
       )
    }
}
```

Figura 82 – Declaração das restrições do estado *PHYUP* da especificação em TDevC utilizada na etapa 3 do primeiro experimento

haja a necessidade de que *driver* solicite ou indique o próximo endereço da memória do *DM9000A*. Por esse motivo, sempre que houver uma escrita neste registrar, este pode se contabilizada como um novo *byte* escrito.

Por fim, outra restrição, chamada de *WriteBeforeLen*, diz respeito ao fato de definir o tamanho do dado escrito antes de escrevê-lo. Esta restrição também pode ser vista na Figura 82. O *datasheet* do controlador *DM9000A* informa que a CPU deve primeiramente escrever o dado na memória do dispositivo (estado *WRITEDATA*) antes de especificar o tamanho do dado escrito (estado *SETLENGTH*).

Uma restrição também foi especificada para o estado *RXPKGHEADERERR*, da região ortogonal *receive*. Esta restrição, chamada de *readInError*, indica que sempre que houver um erro na leitura de um pacote recebido, o *driver* não deve solicitar mais nenhuma byte deste pacote.

Um erro na leitura de um pacote de erro ocorre quando o primeiro byte do cabeçalho de um novo pacote de rede é diferente do valor 0 (zero) ou do valor 1 (um). O valor 0 (zero) indica que nenhum pacote novo se encontra na fila de recebimento de dados e o valor 1 (um) indica que um novo pacote de rede foi recebido e já está pronto para ser lido. A Figura 84 mostra o diagrama de blocos dos pacotes de dados recebidos.

É exatamente com base no formato do pacote de dados recebido que a região ortogonal *receive* está organizada. O estado inicial desta região ortogonal é o estado *WAITNEWDATA*. Neste estado o *driver* fica testando se há algum próximo pacote novo. Enquanto o valor lido for 0 (zero), a execução se mantém no mesmo estado. Caso o valor lido seja 1 (um), a execução passa para o estado *READSTATUS*, caso seja diferente de 1 (um) ou 0

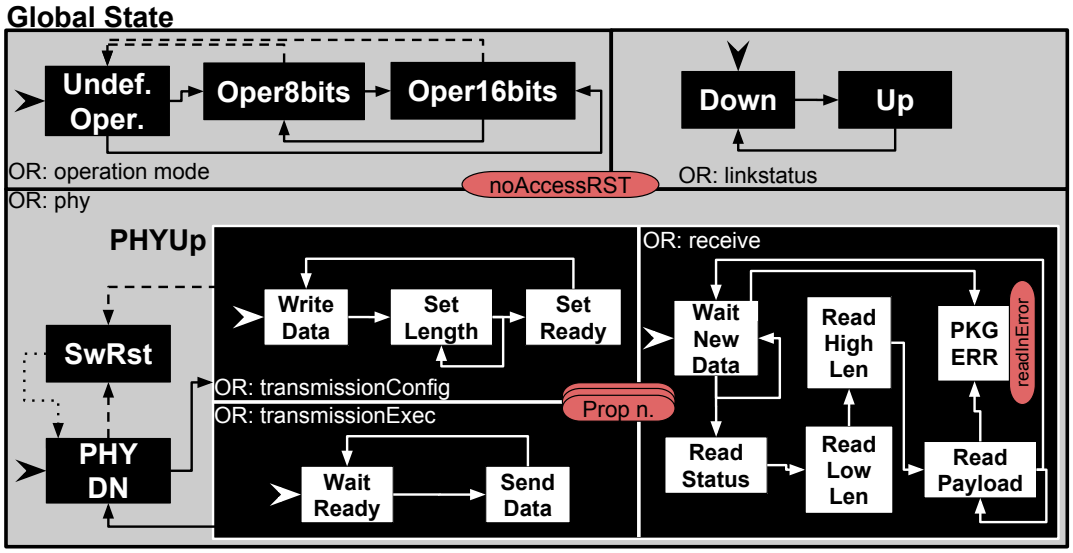


Figura 83 – Representação gráfica da FSM-Monitor da especificação em TDevC utilizada na etapa 3 do primeiro experimento

(zero), a execução vai para o estado de erro *RXPKGHEADERERR*, como mencionando anteriormente.

Após a leitura do status, o *driver* irá solicitar o valor menos significativo do tamanho do pacote, no estado *READLOWLENGTH* e na sequencia o valor mais significativo, no estado *READHIGHLENGTH*. Após isso, o *driver* irá ler todos os dados do *payload* do pacote, no estado *READPAYLOAD*.

Infelizmente a restrição especificada no estado *RXPKGHEADERERR* não pode ser validada, uma vez que não há possibilidade de alterar internamente o controlador *Ethernet* para que ele induza um valor no cabeçalho igual a 2. Contudo, esta restrição foi especificada e o monitor não encontrou nenhuma violação desta restrição do *device driver* do *DM9000A*.

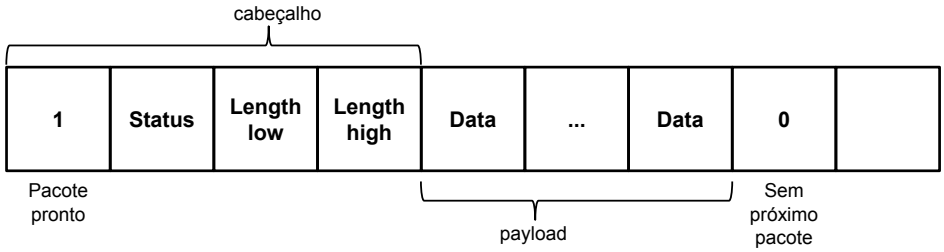


Figura 84 – Divisão do pacote de rede recebido pelo DM9000A

É possível ver graficamente através da Figura 83 a FSM-Monitor que foi especificada em TDevC nas Figuras 79, 81 e 82.

Devido à hierarquia de estados, mesmo com os refinamentos, as propriedades previamente especificadas continuaram valendo para os refinamentos seguintes. As propriedades adicionadas tinham escopos definidos e localizados que permitiram que elas fossem definidas sem precisar alterar ou modificar propriedades gerais anteriormente especificadas. A

propriedade descrita no estado global é automaticamente válida também para os estados filho do estado *PHYUP*, sem que fosse preciso unificar ou referenciar a restrição nestes estados. Isto mostra o suporte da abordagem proposta para o refinamento das propriedades, acompanhando o desenvolvimento gradual do sistema.

É interessante deixar claro que todas as violações mostradas são possíveis de ocorrer, uma vez que não há como restringir isso na programação do protocolo. Não há como bloquear o *device driver* de maneira parcial e momentânea a fazer escritas a endereços limitados do dispositivo. Cabe a ele respeitar a especificação.

Para esta etapa do experimento foi utilizada a ferramenta *UNIX ping* que permite medir a latência da comunicação entre dispositivos de rede. A ferramenta *ping* se baseia no protocolo *Internet Control Message Protocol* (Internet Control Message Protocol (ICMP)), utilizado para trafegar informações sobre a camada de rede entre dispositivos conectados (KUROSE; ROSS, 2007). Porém, para para efetuar essa medição entra as extremidades de uma rota, o pacote ICMP precisa ser recebido por um dispositivo, lido e respondido. Isto é o suficiente para que o *DM9000A* percorra todos os seus estados que estão sendo validados.

Mais uma vez o *driver* original de dispositivo DM9000A foi monitorado e nenhuma violação foi encontrada. Assim, em seguida, ele foi modificado para adicionar violações aleatórias durante a sua execução. Todos os pontos no *device driver* de possíveis violações das propriedades descritas foram identificados e encapsulado com funções que, de maneira randômica, saltavam alguns dos acessos críticos no código do *driver*. Para todo salto forçado executado a função sinalizava a sua ocorrência para que fosse comparada com a detecção da violação por parte do monitor. A técnica proposta detectou todas os erros propositais instantaneamente quando eles ocorreram.

O Figura 85 mostra as alterações no *driver* do *DM9000A* para a indução das três restrições adicionadas no estado *PHYUP*. A primeira modificação, encontrada na Figura 85 entre as linhas 2 e 7, é referente à restrição *WriteBeforeLen*. Este trecho no código força a escrita do tamanho do dado, antes de escrevê-lo na memória do dispositivo.

Como pode ser visto na Figura 86, há dois terminais. O terminal da esquerda é o terminal de um *Linux* sendo executado em uma máquina remota. O terminal da direita é o terminal do μClinux rodando na plataforma sob experimento. Ainda podendo ser visto na Figura 86, foram disparados da máquina remota uma série de requisições ICMP ou *pings* para a plataforma do experimento. Como já mencionado, estas requisições forçam uma resposta do *driver* do controlador *DM9000A*, exigindo que o mesmo execute o trecho de código do *driver* mostrado na Figura 85.

Quando a plataforma responde à requisição ICMP, a indução da restrição ocorre e o monitor a identifica, como pode ser visto na Figura 86, no terminal da direita. Como já comentado, quando ocorre uma violação de uma restrição crítica (tipo *critical*), o monitor suspende a validação do dispositivo. Isso se dá pelo fato de que uma violação de uma restrição crítica pode levar o dispositivo a um estado desconhecido. Por isso, mesmo que

```
1     writeb(DM9000_MWCMD, db->io_addr);
      get_random_bytes(&r, sizeof(unsigned int));
3     if ((r >> 24) < 50){
        printk("\n[DM9000A] TX Pkg Length Setting Failure Induced!\n");
5       iow(db, DM9000_TXPLL, skb->len);
        iow(db, DM9000_TXPLH, skb->len >> 8);
7       }
      get_random_bytes(&r, sizeof(unsigned int));
9     (db->outblk)(db->io_data, skb->data, skb->len);
      dev->stats.tx_bytes += skb->len;
11
      db->tx_pkt_cnt++;
13    /* TX control: First packet immediately send, second packet queue */
      //printk("db->tx_pkt_cnt: %d", db->tx_pkt_cnt);
15    if (db->tx_pkt_cnt == 1) {
       /* Set TX length to DM9000 */
17     if ((r >> 24) > 120){
          printk("\n[DM9000A] TX Pkg Length Value Failure Induced!\n");
19        iow(db, DM9000_TXPLL, skb->len);
          iow(db, DM9000_TXPLH, (skb->len >> 8) + 3);
21     }else{
          iow(db, DM9000_TXPLL, skb->len);
23        iow(db, DM9000_TXPLH, skb->len >> 8);
       }
25        get_random_bytes(&r, sizeof(unsigned int));
          /* Issue TX polling command */
27        iow(db, DM9000_TCR, TCR_TXREQ); /* Cleared after TX complete */
          dev->trans_start = jiffies; /* save the time stamp */
29        if ((r >> 24) < 50){
                  printk("\n[DM9000A] TX Pkg Length Change Failure Induced!\n");
31            iow(db, DM9000_TXPLL, skb->len);
              iow(db, DM9000_TXPLH, (skb->len >> 8) + 5);
33        }
          ...
```

Figura 85 – Trecho da função de transmissão de dados alterada do *device driver* do DM9000A

o *driver* tenha continuado a sua execução após a violação, como mostra o terminal da direita na Figura 86, o monitor não mais a detectou.

É interessante destacar que após a violação, o controlador DM9000A não mais respondeu as requisições de *ping*, como mostram as mensagens "*Destination Host Unreachable*"do terminal da esquerda da Figura 86. Isso mostra que com a violação da restrição, algum ponto da execução do controlador foi para um estado desconhecido.

Em relação à violação da restrição *NeverLenAndSend*, a Figura 87 mostra a execução da plataforma com esta violação induzida. Da mesma maneira que a restrição *WriteBeforeLen* foi testada, a restrição *NeverLenAndSend* também utilizou uma máquina remota, disparando requisições *ping* (terminal da esquerda da Figura 87). A modificação realizada no *driver* que possibilitou a indução desta violação pode ser vista na Figura 85, entre as linhas 29 e 33.

Como pode ser visto no terminal da direita da Figura 87, a violação foi detectada após uma sequencia de *pings* e, do mesmo modo que ocorreu com a violação *WriteBeforeLen*, o monitor desabilita a sua checagem, mesmo ocorrendo mais violações. Para este exemplo é importante destacar que, mesmo ocorrendo a violação da restrição indicada pelo fabricante,

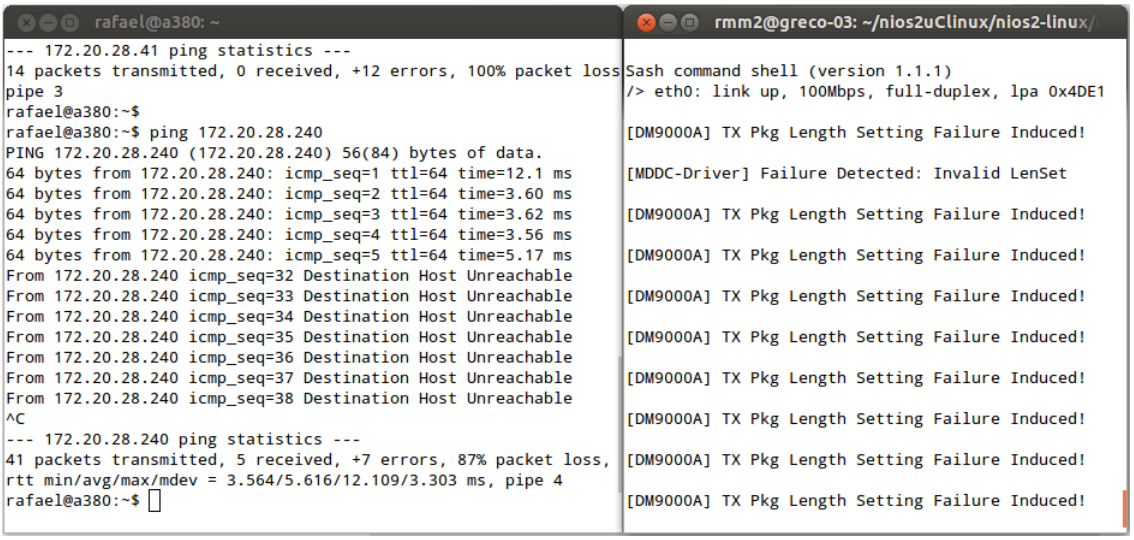


Figura 86 – Terminais *Linux* mostrando o resultado da execução da plataforma com a restrição *LenSet* induzida

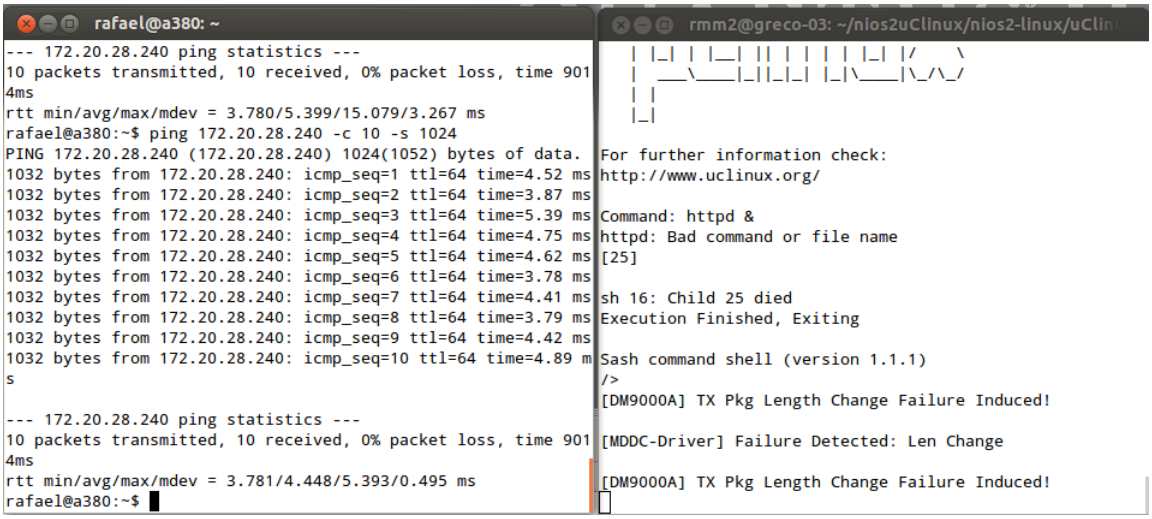


Figura 87 – Terminais *Linux* mostrando o resultado da execução da plataforma com a restrição *Len Change* induzida

o controlador continuou a responder as requisições *ping*. Entretanto, isto não garante que o controlador esteja com o seu funcionamento 100% correto, podendo haver alguma falha que ainda não foi exposta.

Por fim, a Figura 88 mostra o resultado da execução da plataforma com a violação da restrição *TXLenPPT* induzida. Esta restrição diz que sempre que o controlador estiver no estado *SENDDATA*, o valor escrito para o tamanho do dado enviado deve ser (i) exatamente igual ao numero de escritas feitas na memória do controlador, se o modo de operação for de 8-bits, ou (ii) deve ser pelo menos igual ao número de uma escrita a menos que o valor total de escritas realizadas na memória do controlador *DM9000A*, se o modo de operação for de 16-bits.

Estas condições representam respectivamente as seguintes partes da formula LTL

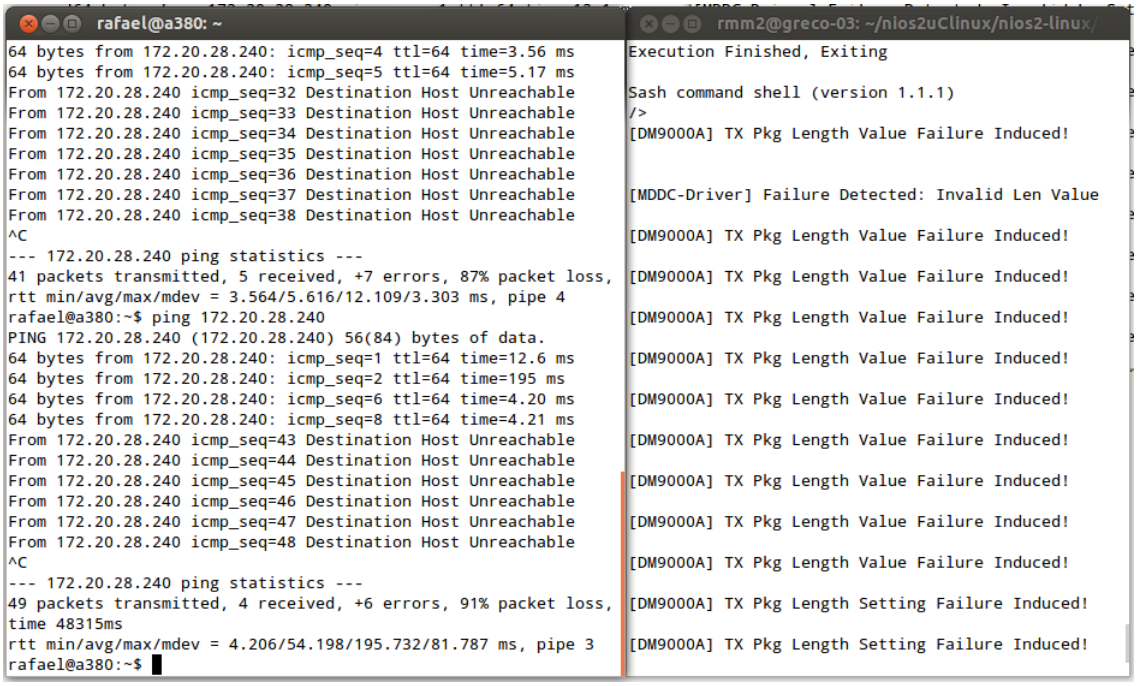


Figura 88 – Terminais *Linux* mostrando o resultado da execução da plataforma com a restrição *Len Value* induzida

mostrada na Figura 82, linha 18: (i) "(*OPER8BITS -> (txlen == txpkgcounter)*"e (ii) "*OPER16BITS -> ((txlen == txpkgcounter) || (txlen ==txpkgcounter-1))*". Isso se dá pelo fato de que quando o controlador está configurado para 16-bits, a cada solicitação de escrita de um dado por parte daCPU, são enviados sempre dois bytes para a memória. Assim, se é solicitado a escrita de um *payload* de tamanho 3 bytes, 4 escritas são realizadas.

Como pode ser visto na Figura 88, o terminal da esquerda envia uma sequencia de requisições *ping* e, na quarta requisição, a violação induzida ocorre. Através do terminal da direita na Figura 88 é possível perceber que o monitor identifica a violação e, como esperado, suspende a checagem, mesmo que ocorram outras violações. Ainda no terminal da direita da Figura 88, é possível ver que uma violação da restrição *WriteBeforeLen*. Mas uma vez que o monitor suspendeu a validação, esta violação não é mais detectada.

A Tabela 2 apresenta mais informações sobre o experimento realizado. A tabela contém o número de linhas de código (Linhas de Código (LdC)) do modelo TDevC, o número de linhas do código gerado na síntese do módulo MDDC, a proporção do código gerado pela síntese em relação à especificação TDevC e o número de estados, registrados e propriedades especificados.

Tabela 2 – Informações sobre a descrição *TDevC* e a síntese

TDevC (LdC)	MDDC (LdC)	Síntese	Estados	Reg.	Propriedades
542	1682	67,7%	20	56	6

Finalmente, a Tabela 3 mostra a área de FPGA utilizada para o experimento, cobrindo

toda a plataforma, o monitor isolado e a relação então a área do monitor e a área da plataforma.

Tabela 3 – Usa de área do FPGA (em elementos lógicos)

Área da plataforma	Área do Monitor	Proporção
7620	870	11.4%

7.2 Experimentos para Medição de Impacto na Execução

Como comentado no início deste capítulo, este experimento tem como objetivo medir o impacto causado na execução do dispositivo quando o *MDDC* está em funcionamento. Através do primeiro experimento deste capítulo foi possível identificar que há impacto no desempenho da plataforma somente quando há um *feedback* da validação para a própria plataforma e, mesmo assim, este impacto ocorre devido à interrupção gerada e tratada pelo *driver* do *MDDC* e não pelo pela presença do monitor. Porém, para deixar confirmado que não há impacto, este experimento irá medir o desempenho do *DM9000A* sob validação para mostrar se há ou não degradação das operações deste dispositivo. Assim, o experimento foi realizado na mesma plataforma do primeiro experimento, com o controlador DM9000A conectado a um computador remoto sem conexão com a internet. O diagrama do ambiente do experimento citado pode ser visto na Figura 89.

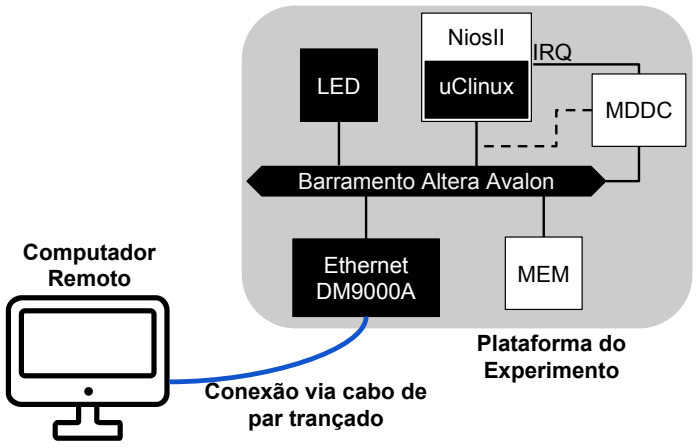


Figura 89 – Arquitetura do ambiente montado para a realização do segundo experimento

Para realização deste dispositivo, executou-se a mesma plataforma do experimento anterior, entretanto com a ferramenta *iperf* (IPERF3, 2018). Ela é uma ferramenta que permite medir a largura de banda e a qualidade do link entre dois pontos. Com ela é possível então avaliar, por exemplo, a latência e a quantidade de datagramas perdidos.

O *iPerf* permite especificar o tamanho do *buffer* a ser lido ou escrito diversas vezes durante um período de tempo que também pode ser especificado. Para os experimentos

foram utilizados tempos de 5 segundos por medição e um tamanho de *buffers* de 1 KB, sendo uma redução do tamanho padrão da ferramenta que é de 10 segundos e 8 KB. Isso se deu pelo fato de que os recursos na plataforma onde o experimento é executado são bem restritos e, com os tamanhos sugeridos pela plataforma, o μClinux embarcado não se mantinha estável, seja com ou sem o *MDDC* na plataforma. Com a redução do tamanho e do tempo, não houve impacto nas medições e ainda manteve a execução estável da plataforma, tornando os resultados mais confiáveis.

O ambiente utilizado neste experimento foi o Kit Altera DE2, com a mesma plataforma do experimento anterior executando em seu FPGA, conectado a um PC com um processador i7 e 8GB de RAM executando um *Linux Ubuntu*. Foram realizadas 80 sequências de execuções utilizando o *iPerf*, sendo que metade destas execuções o PC era o servidor iPerf na outra metade a plataforma era o servidor. Dentre essas 40 vezes para cada um sendo servidor, 20 foram executas com a presença do *MDDC* e 20 sem a presença do mesmo. Assim, são 4 cenários existentes: 20 execuções para o PC servidor com Plataforma cliente **sem** o MDDC, 20 para PC servidor com Plataforma cliente **com** o MDDC, 20 para PC cliente com Plataforma servidor **sem** o MDDC e 20 para PC cliente com Plataforma servidor **com** o MDDC.

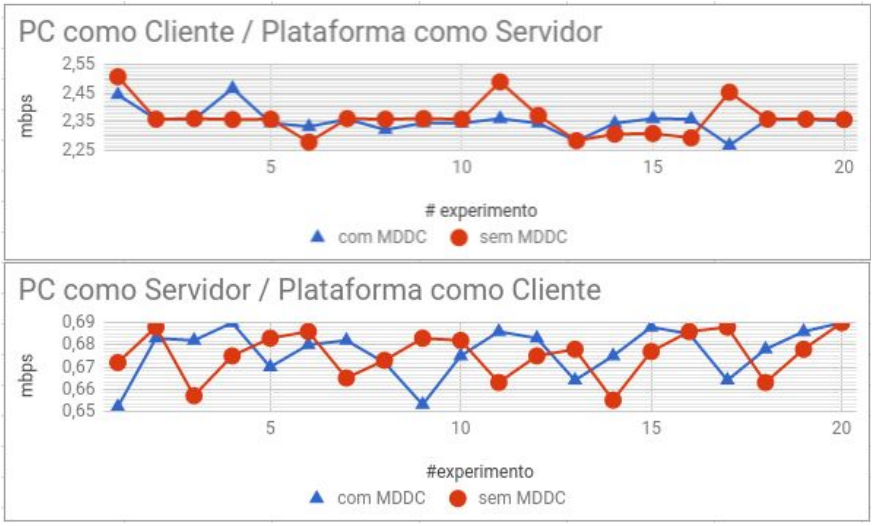


Figura 90 – Gráfico com os dados extraídos do experimento da análise do impacto do MDDC na plataforma

A Figura 90 mostra os resultados obtidos com esse experimento. É possível perceber que as linhas para os experimentos com e sem o *MDDC* variam pelo menos intervalo de maneira aleatória, entretanto quase se sobrepondo, independente de quem seja o servidor ou o cliente. Isso mostra basicamente que não há impacto na execução do dispositivo ou na execução da plataforma quando o *MDDC* está ativo, reforçando o que foi comentado no experimento anterior.

A Tabela 4 mostra os valores discretos obtidos com a execução. É possível perceber que a diferença entra as médias com e sem o *MDDC* é de -0,38% para PC-Cliente/Plataforma-

Tabela 4 – Dados extraídos dos experimentos de largura de banda (em kbps)

#	**PC-Cliente/Plat.-Servidor**		**PC-Servidor/Plat.-Cliente**	
	Com MDDC	**Sem MDDC**	**Com MDDC**	**Sem MDDC**
1	2,444	2,507	0,652	0,672
2	2,359	2,359	0,683	0,688
3	2,359	2,361	0,682	0,657
4	2,466	2,358	0,69	0,675
5	2,346	2,359	0,67	0,683
6	2,333	2,279	0,68	0,686
7	2,359	2,361	0,682	0,665
8	2,322	2,359	0,672	0,673
9	2,346	2,361	0,653	0,683
10	2,345	2,359	0,675	0,682
11	2,361	2,489	0,686	0,663
12	2,345	2,372	0,683	0,675
13	2,282	2,284	0,664	0,678
14	2,345	2,307	0,675	0,655
15	2,361	2,309	0,688	0,677
16	2,359	2,294	0,685	0,686
17	2,268	2,453	0,664	0,688
18	2,358	2,359	0,678	0,663
19	2,359	2,359	0,686	0,678
20	2,353	2,358	0,69	0,69
$\overline{X}$	***2,3535***	***2,3624***	***0,6769***	***0,6759***
σ	***0,0431***	***0,0603***	***0,0113***	***0,0105***
n	***3***	***6***	***1***	***1***

Servidor e +0,16% para PC-Servidor/Plataforma-Cliente, sendo esta diferença uma variação insignificante e comum em tráfego de rede. Isso confirma que não há impacto na execução do dispositivo com a presença do *MDDC*.

A Tabela 4 também mostra que, com base no desvio padrão (σ) dos valores, é possível dizer que, com tamanho da amostra adquirida com experimentos realizados, as médias ($\overline{X}$) representam as populações com um intervalo de confiança de 95%. Para que os resultados obtidos nos experimentos tenham este nível de confiança, é necessário gerar uma quantidade (n) de amostras que represente sua população, ou seja, que represente de maneira confiável o tráfego de rede para aquela situação. Esse nível de confiança está diretamente relacionado ao tamanho da amostra, onde quanto maior o tamanho da amostra maior será sua representatividade e menor será a sua margem de erro). Entretanto,

amostras muito grandes requerem um esforços igualmente grande.

Assim, com base na fórmula (7.1) proposta por Jain (1990), pode-se calcular o tamanho da amostra, para um intervalo de confiança desejado:

$$n = \frac{Z^2 \times \sigma^2}{E_0^2} \tag{7.1}$$

onde Z é o valor crítico que corresponde ao nível de confiabilidade desejada, σ é o desvio padrão populacional e E_0 é o erro amostral tolerável. O erro amostral é comumente definido entre 5% e 10% do valor médio da amostra. Para este caso foi escolhida 5%. Para este intervalo de confiança, o valor de E_0 é definido como 1,967, de acordo com a tabela de referncia fornecida por Lapponi (2004).

7.3 Experimentos Comparativos: *timer* em SCTC

Este último experimento permitiu fazer a comparação entre a abordagem proposta com uma outra técnica que permite a validação de propriedades de sistemas embarcados simulados em plataformas virtuais.

A técnica escolhida para este experimento foi baseada no trabalho de Weiss et al. (2006), e ela possibilita a especificação de propriedades temporais através de um *framework* conhecido como SCTC. O trabalho de Lettnin et al. (2009), discutido no capitulo de trabalhos relacionados (Capítulo 3) deste documento, também faz uso deste *framework* para realizar a validação de propriedades temporais em software embarcado.

Em resumo, o SCTC também se baseia em propriedades temporais descritas através de fórmulas LTL onde suas proposições são sinais lógicos do *framework SystemC* (*sc_signal, sc_in, sc_out, sc_inout*), ou seja, o SCTC também se baseia nos sinais das portas de entrada e saída dos módulos de hardware (modelos SystemC) ou fios (*nets*) internos aos módulos para efetuar a validação. Entretanto, não leva em consideração o protocolo de comunicação de alto nível.

O *framework* fornece um conjunto de funções que permitem a especificação de propriedades LTL e conversão destas fórmulas em máquinas de estados, conhecidas como *AR-Automata* (Automatos de Aceitação e Rejeição). Através da função *sc_monitor* o projetista especifica uma fórmula onde as proposições foram previamente registradas através da declaração de uma classe *C++* herdando a classe *Proposition*.

Para cada proposição se declara uma classe. Com essas classes registradas, as propriedades podem ser declaradas por extenso com o uso da função *sc_monitor*. Por exemplo: *sc_monitor("G (aType & bType)")*, onde *aType* e *tType* foram classes do tipo *Proposition* declaradas previamente e vinculadas a um sinal lógico de *SystemC*.

Para se ter acesso aos sinais, as proposições precisam ser declaradas ou instanciadas no mesmo escopo dos sinais ou estes sinais precisam ser repassados de módulo em módulo (como atributos) até que seja visível pela declaração da proposição. Logo, os módulos

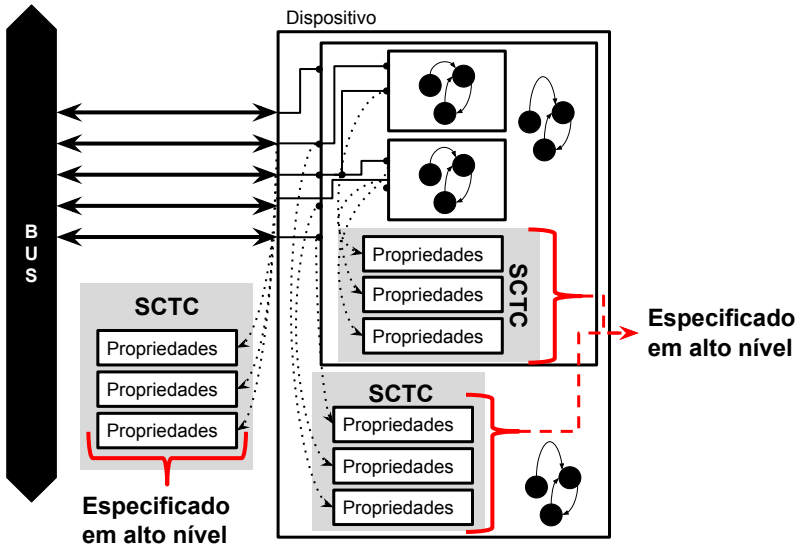


Figura 91 – Aplicação do *framework* SCTC em uma plataforma

precisam ser modificados ou adaptados, sendo assim esta é uma técnica intrusiva e *stateless* em relação ao contexto da execução do dispositivo, uma vez que observa apenas as variações de sinais.

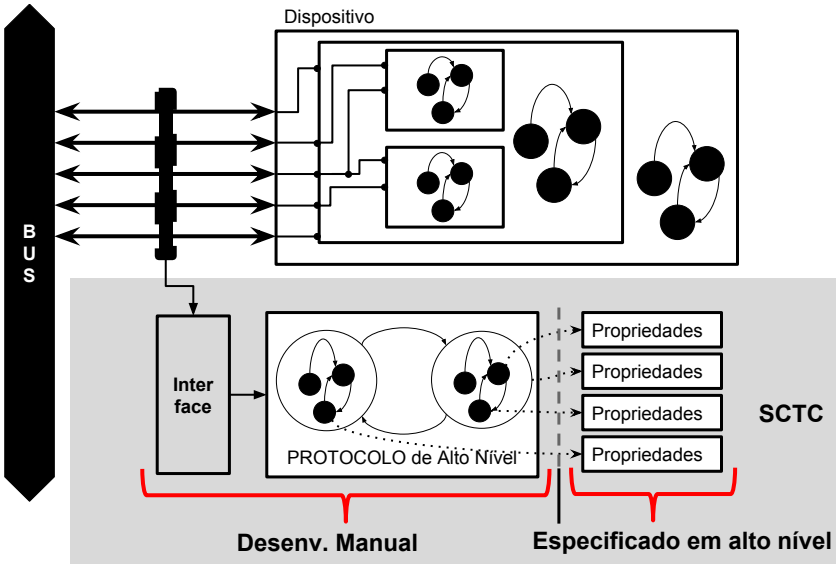


Figura 92 – Aplicação do *framework* SCTC com especificação de estados em uma plataforma

A Figura 91 mostra a técnica sendo aplicada em uma plataforma simples com um dispositivo conectado. Todas as proposições são declaradas manualmente em C++ e as fórmulas são descritas por extenso em alto nível, onde são sintetizadas automaticamente em AR-Automata. A Figura 91 mostra que as propriedades especificadas com o *framework* SCTC ou estão integradas internamente aos módulos do dispositivo sob validação ou os sinais lógicos dos módulos precisam ser expostos para serem acessados externamente, como mostram as setas pontilhadas.

Para se conhecer o estado interno do dispositivo, é preciso construir uma máquina de estados para acompanhar os sinais, funcionalidade não fornecida pela técnica. Dessa maneira essa máquina de estados deve ser construída manualmente em C, C++ ou SystemC, porém, além de ser trabalhoso, o uso de uma linguagem de propósito geral pode acarretar na inserção de erros.

A Figura 92 mostra ainda a técnica proposta pelo *framework* SCTC aplicada com uma máquina de estados que representa o protocolo de alto nível de comunicação com o dispositivo, onda há a representação dos estados do mesmo durante a execução. Isso torna a validação *stateful*, uma vez que agora é possível validar o dispositivo mesmo este sendo uma caixa preta. Para este experimento, foi utilizada a técnica neste formato para que seja mais semelhante à abordagem apresentada neste documento, como pode ser visto na Figura 93.

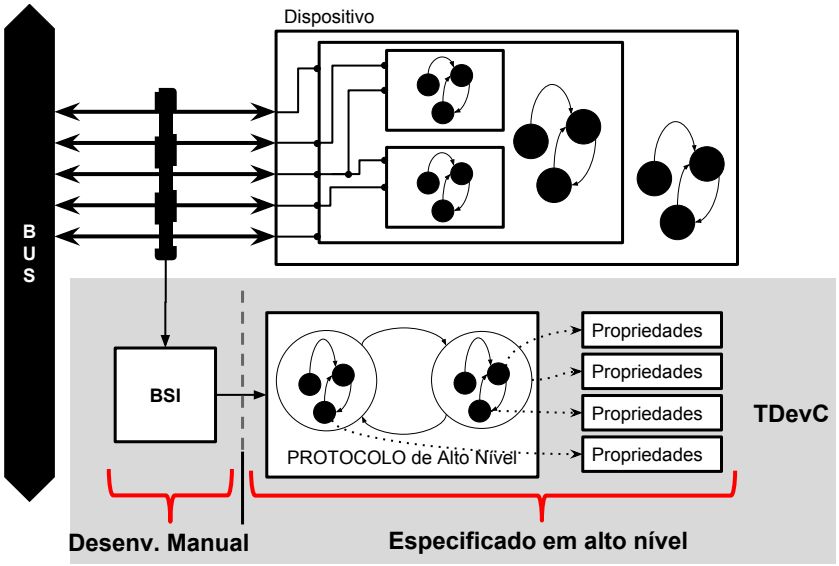


Figura 93 – Aplicação da abordagem proposta em uma plataforma

Quando as duas técnicas são confrontadas, é possível perceber que com a *TDevC* não é possível ter acesso a sinais internos dos dispositivos e esta também não é a intensão. A ideia é ter o conhecimento do estado do dispositivo a partir da sua interface, justamente para não precisar ser intrusivo, porém ter o mínimo possível de programação com uma linguagem do propósito geral.

Outra questão fundamental a ser destacada é o fato de que a técnica com base no SCTC não permite validar plataformas em TLM. Ela exige que os módulos tenhas um *clock* e este *clock* é utilizado também na sincronização dos AR-Automatas. Já com a técnica proposta, a especificação em TDevC pode ser utilizada em nível de transação.

Para este experimento foi utilizado um modelo em *SystemC* de um dispositivo *timer* integrado a uma plataforma TLM contendo um processador *SparcV8* e uma memória de 5 MB. Este dispositivo é simples e apenas duas propriedades foram validadas.

```
globalstate {
    orthoregion onoff{
        initialstate OFF{
            addexitpoint(ON){write(DCR.ON) == 1}
        }
        state ON {
            addexitpoint(OFF){write(DCR.ON) == 0}
            addproperty(critical) noChangeAllowed {
                ltlf( [] (~write(TIMER) && ~write(SCALER)) )
            }
        }
    }
}
```

Figura 94 – Exemplo da descrição da interface de comunicação na linguagem TDevC

O *timer* tem dois estados básicos: *ON* e *OFF*. Além disso, é possível definir um período para o *timer* disparar a interrupção e também é possível definir um *scaler*. O *scaler* é uma espécie de fator multiplicador que faz com que o tempo definido para o *timer* seja multiplique de quantas vezes for definido o valor no *scaler*. Por exemplo, se o valor do *timer* for 10 e o do *scaler* for 10 também, é preciso contar 10 (valor do *timer*) $\times 10 vezes para que a interrupção seja disparada.$

Assim, sempre que o *timer* está desligado, ou seja, no estado *OFF*, é possível definir o valor do *timer* e do *scaler*. Porém, quando o *timer* está ligado (estado *ON*), não é permitido alterar os valores do *timer* ou do *scaler*, tendo como consequência levar o dispositivo a um estado desconhecido.

Foram então especificadas duas restrições para o experimento. Para o framework SCTC as restrições foram escritas da seguinte maneira: *"G !((WRITE(scaler) && ON))"* e *"G !((WRITE(timer) && ON))"*. Elas significam, respectivamente, que sempre (operador *G*), deve ser negado a escrita no registrador *scaler* e estar no estado *ON* e que sempre (operador *G*), deve ser negado a escrita no registrador *timer* e estar no estado *ON*.

Para a especificação TDevC, as restrições foram escritas um pouco diferente porém com o mesmo objetivo. A Figura 94 mostra a FSM-Monitor especificada em TDevC. É possível visualizar na linha 9 da Figura 94 que a restrição especifica apenas que noa é permitido realizar as escritas, uma vez que o fato de estar ou não no estado *ON* já é indiretamente composto com a restrição devido ao fato de que o estado *ON* foi o estado que declarou a restrição.

A especificação em TDevC completa deste experimento pode ser encontrada no Apêndice B.

A especificação apresentada na Figura 94 pode ser vista modela na ferramenta *TDevC-Gen*, juntamente com a representação gráfica dos seus registradores, sua FSM-Monitor e a restrição através da Figura 95 .

A Tabela 5 traz um resumo de como foi a especificação e execução considerando as duas abordagens:

Para a abordagem proposta com o TDevC foram escritas 74 linhas de código, sendo

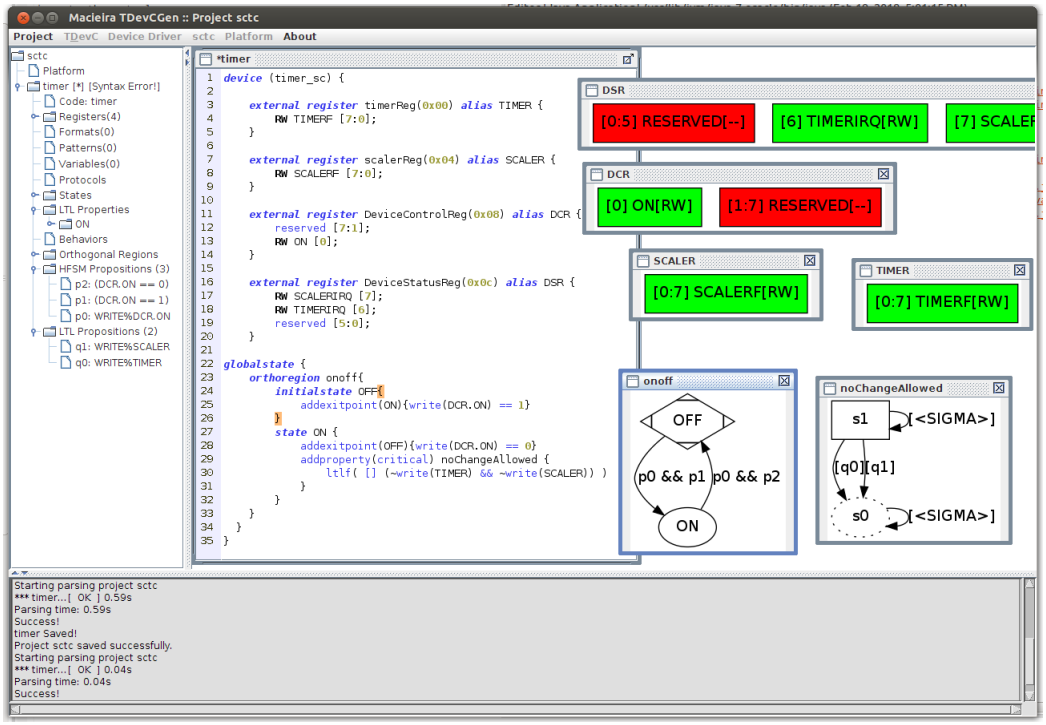


Figura 95 – Aplicação da abordagem proposta em uma plataforma

Tabela 5 – Metricas e valores do experimento comparativo

Metricas	TDevC	SCTC
Linhas de Código Especificadas	74	672
Tempo para especificação	20 min	300 min
Tempo médio de execução	0,378s	0,419s

35 para a especificação em TDevC e 41 para códigos em C necessários para a integração no ambiente, em aproximadamente 20 minutos. Em contrapartida, para o ambiente em SCTC, foram especificadas 672 linhas de código C (aproximadamente 9 vezes maior) em aproximadamente 300 min (15 vezes mais).

É importante destacar que foi preciso fazer uma série de adaptações para que a abordagem baseada em SCTC pudesse ser executada com a plataforma sem precisão de ciclos. Foi necessário criar uma interface que emulasse os ciclos de relógio necessários para executar os AR-Automatas.

Após a especificação das propriedades e plataformas em SCTC e TDevC, foi desenvolvida uma aplicação para usar o *timer*, onde, em o laço de 10000 vezes, o software desliga o *timer*, define o valor do *timer*, do *scaler* e então liga o *timer*. Na última iteração do laço o software não desliga o *timer* antes de definir o valor e *scaler*.

Para este experimento, a aplicação foi executada 10 vezes para cada ambiente. A Figura 96 mostra o desempenho para cada um dos ambientes. É possível perceber que o tempo de execução para o TDevC com o MDDC foi levemente menor do que o tempo com o SCTC em aproximadamente **9,8%**. Isso se dá pelo fato de que o MDDC é apenas um chamada de

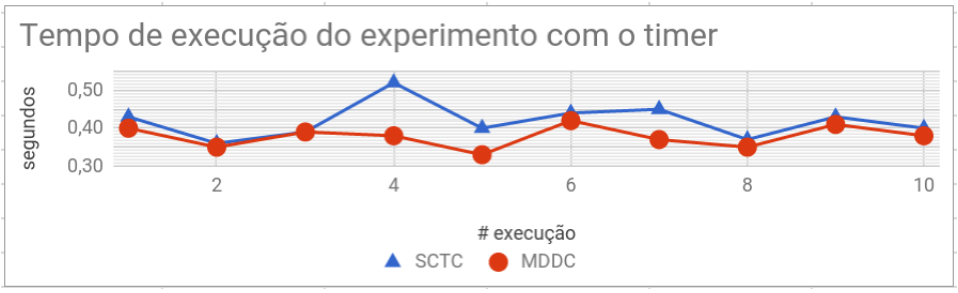


Figura 96 – Tela da ferramenta TDevCGen mostrando a especificação em TDevC utilizada para o terceiro experimento

Tabela 6 – Dados extraídos dos experimentos comparativo

#	SCTC	TdevC
1	0,43	0,4
2	0,36	0,35
3	0,39	0,39
4	0,52	0,38
5	0,4	0,33
6	0,44	0,42
7	0,45	0,37
8	0,37	0,35
9	0,43	0,41
10	0,4	0,38
Media	***0,4190***	***0,3780***
desvio	***0,0463***	***0,0286***
amostras	***4***	***1***

função dentro da plataforma. Em contrapartida, o SCTC requisitou um módulo SystemaC com controle de ciclo de relógio para que os dados fossem validados, contando assim com uma troca de contexto de threads no *kernel* do SystemC.

A Figura 6 mostra os dados discretos das execuções dos dois ambientes. No final da tabela, assim como no experimento anterior, contém a média dos dados, o desvio padrão e o tamanho mínimo de amostra para representar a população.

7.4 Resumo

Este capítulo apresenta os resultados obtidos com os experimentos realizados. O primeiro experimento validou a viabilidade e eficácia da abordagem. O segundo experimento validou o impacto na execução que o monitor traz à plataforma. Já o terceiro experimento fez um estudo comparativo com a técnica baseada no SystemC Temporal Checker (SCTC) e

confronta tanto o tempo de execução como a produtividade das duas técnicas.

8 CONCLUSÃO

Este trabalho mostrou a necessidade de se desenvolver *device drivers* seguros e mostrou também a lacuna que as técnicas e metodologias atuais de projetos de sistemas têm em relação à validação da comunicação entre elementos de processamento de uma plataforma.

Foi visto também que muitos destes sistemas, como por exemplo CPS, são altamente críticos, o que implica que uma falha no controle de um atuador ou na leitura de sensor pode provocar acidentes catastróficos, envolvendo até vidas humanas.

Foi destacada a importância de se ter um formalismo de especificações nos projetos distribuídos de componentes de sistemas computacionais. Além da especificação, a checagem de suas características e propriedades é fundamental para se manter a qualidade do sistema, porém, deve suportar tanto as fase de desenvolvimento quanto sua fase de operação. Esta segunda bastante importante nos dispositivos atuais, uma vez que a maioria deles está conectada à internet quase que em tempo integral e por isso são bastante susceptíveis a invasões e consequentemente a alterações nos seus comportamentos.

Algumas abordagens tratam com sucesso problemas relacionados, propondo modelos em alto nível para especificar propriedades de componentes através da ideia de contratos. Entretanto seus modelos ou são muito genéricos ou focados em interfaces físicas de componentes, o que torna bastante complicada a especificação de protocolos de comunicação em alto nível, ou são orientadas à implementação da pilha de software, o que as torna dependentes de códigos fontes e sensíveis a códigos de terceiros. Esta dependência torna inviável a utilização destas técnicas na checagem de sistema durante a fase de operação, uma vez que durante esta fase pode haver diversas atualizações na camada de software.

Assim, este trabalho desenvolveu uma abordagem que permitisse especificar modelos em alto nível que representassem os protocolos de comunicação entre processadores e periféricos de plataformas e propriedades relacionadas a execução deste protocolo, na forma de contratos, e que estes modelos auxiliassem o desenvolvimento incremental do sistema, sendo também aproveitados durante a sua fase de operação.

Para isso foram desenvolvidas uma linguagem de especificação de modelos chamada *TDevC* e uma técnica de síntese desses modelos implementada através da uma ferramenta chamada *TDevCGen*. Experimentos realizados com a abordagem proposta mostraram a sua viabilidade e sua eficácia na detecção das violações das propriedades especificadas.

Porém, além de mostrar e sua viabilidade de uso e sua eficácia em detectar violações, através dos experimentos foi possível ver que a abordagem realmente se enquadra em uma metodologia incremental de desenvolvimento de sistemas. O suporte à descrição de protocolos de maneira hierárquica por parte da linguagem *TDevC* se mostrou ideal para se incorporar mais detalhes tanto em relação ao protocolo em si, como em relação às suas propriedades. Os experimentos relacionados também deixaram claro que é possível

adicionar mais detalhes sem que características prévias das descrições fossem modificadas ou removidas.

Dois outros pontos expostos pelos experimentos são relacionados ao impacto no desempenho da execução do sistema quando o MDDC está integrado à plataforma e a produtividade da abordagem em relação a uma técnica similar. Estes experimentos mostraram que o MDDC não gera um impacto na execução do sistema.

Em relação à produtividade no desenvolvimento de um ambiente de validade de HdS, esta abordagem se mostrou bastante eficiente quando comparada com outra técnica que propõe a validação de propriedades em sistemas. Isso se deu pelo fato de que a linguagem TDevC, por ser uma DSL, abstrai bastante informações referentes à montagem de mensagens na interface dos dispositivos, fazendo que o sintetizador gere algumas especificações seriam de responsabilidade do projetista, quando utilizada uma linguagem de propósito geral.

Entretanto, mesmo possibilitando o incremento da descrição, a sua validação prévia durante a síntese também se mostrou fundamental. Não adianta adicionar detalhes e propriedades a uma descrição se estes irão inviabilizar os que já tinha sido anteriormente descrito.

Outro atributo que se mostrou fundamental foi a existência das regiões ortogonais. Através delas foi possível descrever contextos ou modos de operações concorrentes. Com o experimento envolvendo o dispositivo *Ethernet*, este atributo ficou evidente. Mesmo que a configuração, escrita e leitura de dados durante a transmissão e a recepção de pacotes sejam feitas de maneira sequencial, há uma espécie de chaveamento entre o protocolo relacionando à transmissão e o relacionado à recepção de dados da *Ethernet*. Só foi possível descrever este comportamento devido à existência desses mecanismos de especificação de modos de operações ou linhas de execução concorrentes suportados pela linguagem *TDevC*.

Uma grande vantagem na aplicação desta técnica para detecção de violações de propriedades em sistemas embarcadas se deu pelo fato de que todo o seu monitoramento é completamente independente da pilha de software em execução na plataforma. Os modelos descritos em *TDevC* são puramente baseados nas especificações dos dispositivos, sendo completamente desvinculados de nomes e endereços de variáveis e endereços de funções de software. A grande vantagem disto está no fato de que esta técnica pode ser aplicada durante a fase de operação do sistema, sendo completamente insensível à variação de qualquer componente na camada de software, seja o próprio *device driver* ou qualquer software de terceiros. Justamente por ser completamente independente do software, esta abordagem não necessita de nenhuma anotação e modificação de componentes de software para o seu funcionamento. Isso se mostrou uma vantagem uma vez que todas as características da pilha de software se mantêm intactas, preservando assim suas características de execução como o consumo de memória e propriedades temporais.

Além de preservar as características temporais da execução do software, esta abordagem mostrou que nenhuma característica temporal da plataforma foi prejudicada. Sendo completamente não intrusiva na execução do sistema, os únicos *overhead* gerados foram relacionados ao consumo de energia e área de FPGA, devido ao hardware adicional nas plataformas reais, e o aumento no tempo de simulação e consumo de memória para as plataformas virtuais.

Apesar dos experimentos cobrirem a maior parte dos objetivos propostos por esta abordagem, a checagem de propriedades com restrições temporais foi realizada parcialmente. Entretanto, apesar da abordagem ainda não dar suporte completamente a esta modalidades de checagem, experimentos realizados manualmente já validaram a viabilidade, como já comentado no Capítulo 7.

Contudo, a abordagem tem algumas limitações. Uma delas é o fato de que, apesar de ser possível realizar o monitoramento de interfaces de componentes de hardware, o controle destes dispositivos deve ser realizado através da escrita e leitura de registradores, uma vez que a linguagem *TDevC* é fundamentada neste modelo de comunicação.

Uma limitação está também no monitoramento de dados escritos em memória principal. O único tipo de elementos de memória suportado pela técnica é o registrador. Assim, é interessante definir uma elemento do tipo memória, onde possa ser possível definir o tamanho maior que o tamanho da palavra do barramento (como atualmente é definido o tamanho dos registradores na linguagem *TDevC*) e com faixa de endereçamento. Com a declaração de um tipo de dados de memória, é possível definir na *FSM-Monitor* um padrão de acesso à esta memória e com isso potencializar o uso da técnica, permitindo a sua aplicação com dispositivos como, por exemplo, o *DMA*.

Outra limitação é que a linguagem não suporta a especificação do protocolo do barramento ou elemento de comunicação envolvido no monitoramento. Atualmente o módulo *BSPI* é um componente extraído de uma biblioteca contendo implementações prévias do protocolo de comunicação dos elemento de comunicação envolvidos. Assim, caso um novo elemento de comunicação seja utilizado, o *BSPI* deverá ser implementado manualmente. É interessante que a *TDevC* dê suporte à especificação deste protocolo e sintetize automaticamente o módulo *BSPI*.

Concluindo, a Tabela 7 retoma o comparativo entre os trabalhos relacionados neste documento, só que desta vez adicionando esta abordagem na comparação. Das 13 métricas avaliadas, 11 foram atendidas completamente, 1 de maneira parcial e 1 não foi contemplada.

Logo, esta abordagem se mostrou a única que deu suporte à checagem de propriedades de protocolo de alto nível de comunicação entre componentes de uma plataforma embarcadas, tanto na sua fase de desenvolvimento quanto na fase de operação, de maneira não intrusiva e totalmente independente da implementação do código do software embarcado. Além disso, ela permitiu a descrição e validação das propriedades de maneira incremental, em concordância com as metodologias de desenvolvimento de sistemas atuais.

Tabela 7 – Comparação entre os trabalhos relacionados e a abordagem proposta

Caract. / Aborda.	Suporta						Indeped.		(9) Insensib. a libs de SW de terceiros	(10) Exec. não Intr.	(11) Sem Anot. de Código	(12) Val. das Propr.	(13) Conc.
	(1) HdS	(2) Fase Desenv.	(3) Desenv. Increm.	(4) Fase Oper.	(5) Prop. Modo Oper/ Estado	(6) Restr. temp.	(7) Bin.	(8) Cód. Fonte					
Cout. '06	×	■	×	□	■	×	×	×	×	×	■	×	×
Pelizz. '08	■	■	×	■	□	×	■	■	■	□	■	×	×
Lettnin'09	□	■	×	×	□	□	□	×	×	×	×	×	□
Macieira'11	■	■	×	×	□	×	×	×	×	■	■	×	×
Reinb. '12	×	■	×	×	×	■	×	×	×	■	■	×	×
Nuzzo '14	■	■	■	×	×	■	■	■	■	■	■	□	×
Ferrante'14	■	■	■	×	■	■	■	■	■	■	■	×	□
Zheng '15	×	■	×	■	■	■	×	×	×	×	×	×	□
Decker '18	■	■	×	□	×	■	×	×	×	■	■	×	□
***Macieira**	■	■	■	■	■	□	■	■	■	■	■	■	×

Suporta	■
S. Parcial	□
N. Suporta	×
Proposta	*

Porém, algumas pendências ainda precisam ser resolvidas. A seção a seguir irá detalhá-las.

8.1 Trabalhos Futuros

Características do protocolo e propriedades relacionadas a restrições temporais ainda não são suportadas completamente nem pela linguagem e nem pela ferramenta. Porém, apesar disso, alguns testes manuais foram realizados e foi comprovada a viabilidade de se ter propriedades e transições de protocolos com base em restrições temporais.

Outra questão que fará com que a técnica suporte mais fortemente o desenvolvimento de sistemas modernos é a sua aplicabilidade com abrangência em concorrência de acesso a dispositivos. Atualmente esta técnica não suporta monitoramento de acessos concorrentes a dispositivos, seja por sistemas *multithread* ou *multicores*. A definição de seções críticas na especificação do protocolo de comunicação para acessos concorrentes e a existência de MDDC locais para elemento de processamento faz com que a técnica tenha controle de múltiplos acessos, identificando situações em que diferentes *threads* mudem configurações de dispositivos sem que outra *thread* tenha conhecimento, violando o acesso à seção crítica.

A definição de ações ou transformações de dados vinculadas às definições das propriedades é algo também que merece um estudo mais aprofundado. Algumas técnicas discutidas na seção de Trabalhos Relacionados deste documento propõem mecanismos reativos, tomando uma ação quando uma falha ou violação é encontrada. A TDevC não tem nenhuma estrutura que permita especificar uma ação. Além disso, por não ser intrusivo, o MDDC atualmente não teria como realizar alguma ação diante da violação. Desta maneira, seria necessário acrescentar na gramática da linguagem uma maneira de definir ação e associá-la a uma propriedade, além de que o MDDC necessitaria ser modificado para, ao invés de observar a comunicação, ele interceptar e repassar os dados das mensagens trocadas entro o elemento de processamento e os dispositivos.

Em relação à validação em plataformas virtuais, o experimento comparativo com a técnica apresentada por Lettnin et al. (2009) mostrou ser interessante e bastante promissora na exploração de uma integração da abordagem proposta neste documento com o SCTC, uma vez que este *framework* foi amplamente validado com a indústria e tem bastante suporte e ferramentas que apoiam o seu uso.

Outra questão bastante interessante em relação ao trabalho apresentado por Lettnin et al. (2009) é o uso da FLTL. Seria bastante enriquecedor que a TDevC suportasse esta linguagem lógica para que fosse possível a especificação de propriedades temporais com informações de tempo. Assim, a TDevC poderá dar suporte a especificação de propriedades com restrições de tempo.

Um ponto importante a ser tratado futuramente é o desenvolvimento de uma meta especificação também baseada em TDevC. Atualmente todos os dispositivos são monitorados de maneira isolada. Com uma meta especificação seria possível monitorar a relação

entre todos os dispositivos, expandido o alcance de visibilidade da TDevC em relação à pilha de software. Desta maneira seria possível modelar contextos de aplicação do sistema, tendo uma visão além do controle isolado do dispositivo.

Permitir a análise da cobertura também seria uma funcionalidade que agregaria bastante valor à abordagem. O trabalha proposto por Macieira, Lisboa e Barros (2011) permitia a visualização de um relatório final após a execução da validação, informando quais serviços dos *device drivers* haviam sido consumidos e quais registradores haviam sido utilizados. Para esta abordagem atual fica a sugestão da implementação de um mecanismo que contabilize quais estados e caminhos do *trace* foram estressados e quantas vezes.

Uma outra funcionalidade que contribuiria bastante com a natureza BIST do *MDDC* seria a realização de auto testes em momentos distintos: (i) após a sua geração, onde, jutamente com uma nível de cobertura definido, este teste serviria como um *testbench* para garantir a corretude por construção do monitor, e (ii) em tempos esporádicos para validar que não houve alteração na lógico do monitor por influências físicas externas, sejam elas radiações, envelhecimento, interferência mecânica, etc.

Por fim, é fundamental aprimorar os teste e experimentos realizados com esta abordagem. Primeiramente e um dos principais trabalhos futuros, seria a aplicação desta abordagem em um estudo de caso real. Com a aplicação da técnica em um ambiente real, todos os requisitos e necessidade para uso de uma abordagem desta natureza se tornariam muito mais claros e palpáveis.

Entretanto, mesmo em um ambiente controlado e de laboratório, o aprimoramento dos teste e experimentos pode ser realizado através da aplicação da técnica em diversos e diversificados dispositivos e/ou categorias de dispositivos, como por exemplo, módulos WIFI, módulos de controle de memória secundária, módulos controle mecânico, etc.

Outra maneira de realizar experimentos e testes mais conclusivos é através do uso teste de mutação ou o uso de versões mais antigas e com erros conhecidos de *device drivers*, começando inclusivo com versões mais antigas do *driver* do controlador Ethernet DM9000A. Partido desta abordagem é possível contar com um banco de *bugs* conhecidos para mensurar o poder de detecção de erros da técnica.

REFERÊNCIAS

ACQUAVIVA, A.; BOMBIERI, N.; FUMMI, F.; VINCO, S. Semi-automatic generation of device drivers for rapid embedded platform development. *IEEE Trans. on CAD of Integrated Circuits and Systems*, v. 32, n. 9, p. 1293–1306, 2013. Disponível em: <http://dblp.uni-trier.de/db/journals/tcad/tcad32.html#AcquavivaBFV13>.

ARM Keil Microcontroller Tools. 2016. <http://www.keil.com/>. Accessed: 2016-03-29.

AVIZIENIS, A.; LAPRIE, J.-C.; RANDELL, B.; LANDWEHR, C. Basic concepts and taxonomy of dependable and secure computing. *IEEE Trans. Dependable Secur. Comput.*, IEEE Computer Society Press, Los Alamitos, CA, USA, v. 1, n. 1, p. 11–33, jan. 2004. ISSN 1545-5971. Disponível em: <http://dx.doi.org/10.1109/TDSC.2004.2>.

BAIER, C.; KATOEN, J.-P. *Principles of Model Checking (Representation and Mind Series)*. [S.l.]: The MIT Press, 2008. ISBN 026202649X, 9780262026499.

BALASUBRAMANIAN, D.; LOWRY, M.; CORINA, P. Rapid Property Specification and Checking for Model-based Formalisms. p. 121–127, 2011.

BEHREND, J.; GRUENHAGE, A.; SCHROEDER, D.; LETTNIN, D.; RUF, J.; KROPF, T.; ROSENSTIEL, W. Optimized hybrid verification of embedded software. In: *Test Workshop - LATW, 2014 15th Latin American*. [S.l.: s.n.], 2014. p. 1–6.

BEHREND, J.; LETTNIN, D.; HECKELER, P.; RUF, J.; KROPF, T.; ROSENSTIEL, W. Scalable hybrid verification for embedded software. In: *DATE*. IEEE, 2011. p. 179–184. ISBN 978-1-61284-208-0. Disponível em: <http://dblp.uni-trier.de/db/conf/date/date2011.html#BehrendLHRKR11>.

BENVENISTE, A.; DAMM, W.; SANGIOVANNI-VINCENTELLI, A.; NICKOVIC, D.; PASSERONE, R.; REINKEMEIER, P. Contracts for the design of embedded systems part i: Methodology and use cases. *Contract*, v. 2, p. G1, 2011.

BEUCHER, O. *MATLAB and Simulink (Scientific Computing)*. [S.l.]: Pearson Studium, 2006. ISBN 3827372062.

BLACK, D. C.; DONOVAN, J.; BUNTON, B.; KEIST, A. *SystemC: From the ground up*. [S.l.]: Springer Science & Business Media, 2009. v. 71.

BOMBIERI, N.; FUMMI, F.; PRAVADELLI, G.; VINCO, S. Correct-by-construction generation of device drivers based on RTL testbenches. *2009 Design, Automation & Test in Europe Conference & Exhibition*, p. 1500–1505, 2009. ISSN 1530-1591. Disponível em: <http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=5090900>.

CAI, L.; GAJSKI, D. Transaction level modeling: an overview. In: ACM. *Proceedings of the 1st IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis*. [S.l.], 2003. p. 19–24.

CARVALHO, V. L. A. de. *VALIDAÇÃO DE UMA ESPECIFICAÇÃO TDEVC PARA O DESENVOLVIMENTO DE DEVICE DRIVERS ROBUSTOS*. Dissertação (Mestrado), 2016.

CHEN, F.; ROsU, G. Mop: An efficient and generic runtime verification framework. *SIGPLAN Not.*, ACM, New York, NY, USA, v. 42, n. 10, p. 569–588, out. 2007. ISSN 0362-1340. Disponível em: <http://doi.acm.org/10.1145/1297105.1297069>.

CHOU, A.; YANG, J.; CHELF, B.; HALLEM, S.; ENGLER, D. *An empirical study of operating systems errors*. [S.l.]: ACM, 2001. v. 35.

CLARKE, E. M.; GRUMBERG, O.; PELED, D. *Model checking*. [S.l.]: MIT press, 1999.

CONWAY, C. L.; EDWARDS, S. A. NDL: A Domain-Specific Language for Device Drivers. p. 30–36, 2004.

COUTINHO, A. E. V. B. *Detecção Automática de Violações de Propriedades de Sistemas Concorrentes em Tempo de Execução*. Dissertação (Mestrado), 2006.

DAMM, W.; HUNGAR, H.; JOSKO, B.; PEIKENKAMP, T.; STIERAND, I. Using contract-based component specifications for virtual integration testing and architecture design. In: IEEE. *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2011*. [S.l.], 2011. p. 1–6.

DAVICOM. *DM9000A 16/8 Bit Ethernet Controller with General Processor Interface - Application Notes V1.20*. [S.l.], 2005.

DAVICOM. *DM9000A Ethernet Controller with General Processor Interface - Datasheet*. [S.l.], 2006.

DECKER, N.; DREYER, B.; GOTTSCHLING, P.; HOCHBERGER, C.; LANGE, A.; LEUCKER, M.; SCHEFFEL, T.; WEGENER, S.; WEISS, A. Online analysis of debug trace data for embedded systems. In: *2018 Design, Automation Test in Europe Conference Exhibition (DATE)*. [S.l.: s.n.], 2018. p. 851–856.

D'ANGELO, M. *Software Platform for Integration of Engineering and Things*. [S.l.], 2014.

ECKER, W.; MUELLER, W.; DOMER, R. *Hardware-depended Software: Principles and Pratice*. [S.l.: s.n.], 2009.

EN, N. C. 50129: Railway application–communications, signaling and processing systems–safety related electronic systems for signaling. *British Standards*, 2003.

FERRANTE, O.; PASSERONE, R.; FERRARI, A.; MANGERUCA, L.; SOFRONIS, C. Bcl: A compositional contract language for embedded systems. In: *Emerging Technology and Factory Automation (ETFA), 2014 IEEE*. [S.l.: s.n.], 2014. p. 1–6.

FERRANTE, O.; PASSERONE, R.; FERRARI, A.; MANGERUCA, L.; SOFRONIS, C.; D'ANGELO, M. Monitor-based run-time contract verification of distributed systems. *Proceedings of the 9th IEEE International Symposium on Industrial Embedded Systems, SIES 2014*, 2014.

FERREIRA, A. B. d. H. *Novo dicionário da língua portuguesa*. [S.l.]: Nova Fronteira, 1986.

FERREIRA, R. S. *Stack smashing attack detection methodology for secure program execution based on hardware*. Dissertação (Mestrado) — Pontifícia Universidade Católica do Rio Grande do Sul, 2016.

FERREIRA, R. S.; VARGAS, F. Shadowstack: A new approach for secure program execution. *Microelectronics Reliability*, Elsevier, v. 55, n. 9, p. 2077–2081, 2015.

GAJSKI, D. D.; ABDI, S.; GERSTLAUER, A.; SCHIRNER, G. *Embedded System Design: Modeling, Synthesis and Verification*. 1st. ed. [S.l.]: Springer Publishing Company, Incorporated, 2009. ISBN 1441905030, 9781441905031.

GAJSKI, D. D.; VAHID, F.; NARAYAN, S.; GONG, J. *Specification and design of embedded systems*. [S.l.]: PTR Prentice Hall Englewood Cliffs, New Jesey, USA, 1994.

GANAPATHI, A.; GANAPATHI, V.; PATTERSON, D. Windows xp kernel crash analysis. In: *Proceedings of the 20th Conference on Large Installation System Administration*. Berkeley, CA, USA: USENIX Association, 2006. (LISA '06), p. 12–12. Disponível em: <http://dl.acm.org/citation.cfm?id=1267793.1267805>.

GANAPATHY, V.; BALAKRISHNAN, A.; SWIFT, M. M.; JHA, S. Microdrivers: A new architecture for device drivers. In: HUNT, G. C. (Ed.). *HotOS*. USENIX Association, 2007. Disponível em: <http://dblp.uni-trier.de/db/conf/hotos/hotos2007.html#GanapathyBSJ07>.

GASTIN, P.; ODDOUX, D. Fast ltl to büchi automata translation. In: BERRY, G.; COMON, H.; FINKEL, A. (Ed.). *Computer Aided Verification*. Springer Berlin Heidelberg, 2001, (Lecture Notes in Computer Science, v. 2102). p. 53–65. ISBN 978-3-540-42345-4. Disponível em: <http://dx.doi.org/10.1007/3-540-44585-4_6>.

GHENASSIA, F. et al. *Transaction-level modeling with SystemC*. [S.l.]: Springer, 2005.

GROBMEIER, C. *The New Log4J 2.0*. [S.l.], 2012. Disponível em: <https://www.grobmeier.de/the-new-log4j-2-0-05122012.html>.

HAREL, D. Statecharts: A visual formalism for complex systems. *Science of computer programming*, Elsevier, v. 8, n. 3, p. 231–274, 1987.

HERDER, J. N.; BOS, H.; GRAS, B.; HOMBURG, P.; TANENBAUM, A. S. Failure resilience for device drivers. In: *Proceedings of the 37th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*. Washington, DC, USA: IEEE Computer Society, 2007. (DSN '07), p. 41–50. ISBN 0-7695-2855-4. Disponível em: <http://dx.doi.org/10.1109/DSN.2007.46>.

HOOLE, A. M.; TRAORE, I. Contract-based security monitors for service oriented software architecture. *Proceedings of the 3rd IEEE Asia-Pacific Services Computing Conference, APSCC 2008*, p. 1239–1245, 2008.

HUANG, H.-W. *Using the MCS-51 microcontroller*. [S.l.]: Oxford University Press, Inc., 2000.

IPERF3. 2018. <https://iperf.fr/iperf-doc.php>. Accessed: 2018-01-03.

ITRS Report. [S.l.], 2015. Disponível em: <http://www.itrs2.net/>.

JAIN, R. *The art of computer systems performance analysis: techniques for experimental design, measurement, simulation, and modeling*. [S.l.]: John Wiley & Sons, 1990.

JANKA, R. S. *Specification and design methodology for real-time embedded systems*. [S.l.]: Springer Science & Business Media, 2002.

KADAV, A.; RENZELMANN, M. J.; SWIFT, M. M. Tolerating hardware device failures in software. In: *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles*. New York, NY, USA: ACM, 2009. (SOSP '09), p. 59–72. ISBN 978-1-60558-752-3. Disponível em: <http://doi.acm.org/10.1145/1629575.1629582>.

KATAYAMA, T.; SAISHO, K.; FUKUDA, A. Prototype of the device driver generation system for unix-like operating systems. In: *Principles of Software Evolution, 2000. Proceedings. International Symposium on*. [S.l.: s.n.], 2000. p. 302–310.

KICZALES, G.; HILSDALE, E.; HUGUNIN, J.; KERSTEN, M.; PALM, J.; GRISWOLD, W. G. An overview of aspectj. In: *Proceedings of the 15th European Conference on Object-Oriented Programming*. London, UK, UK: Springer-Verlag, 2001. (ECOOP '01), p. 327–353. ISBN 3-540-42206-4. Disponível em: <http://dl.acm.org/citation.cfm?id=646158.680006>.

KNIGHT, J. *Fundamentals of Dependable Computing for Software Engineers*. 1st. ed. [S.l.]: Chapman & Hall/CRC, 2012. ISBN 1439862559, 9781439862551.

KRöGER, F. *Temporal Logic of Programs*. [S.l.]: Springer-Verlag, 1987.

KUROSE, J.; ROSS, K. *Redes de computadores e a Internet: uma abordagem top-down*. ADDISON WESLEY BRA, 2007. ISBN 9788588639188. Disponível em: <https://books.google.com.br/books?id=i5WwAAAACAAJ>.

LAPPONI, J. C. *Estatística usando excel*. [S.l.]: Elsevier Brasil, 2004.

LARMAN, C.; BASILI, V. R. Iterative and incremental development: A brief history. *Computer*, IEEE, n. 6, p. 47–56, 2003.

LEE, E. A. *Cyber Physical Systems: Design Challenges*. [S.l.], 2008. Disponível em: <http://www.eecs.berkeley.edu/Pubs/TechRpts/2008/EECS-2008-8.html>.

LETTNIN, D. *Verification of Temporal Properties in Embedded Software*. Tese (Doutorado) — Universität Tübingen, 2009.

LETTNIN, D.; NALLA, P.; BEHREND, J.; RUF, J.; GERLACH, J.; KROPF, T.; ROSENSTIEL, W.; SCHONKNECHT, V.; REITEMEYER, S. Semiformal verification of temporal properties in automotive hardware dependent software. In: *Design, Automation Test in Europe Conference Exhibition, 2009. DATE '09.* [S.l.: s.n.], 2009. p. 1214–1217. ISSN 1530-1591.

LETTNIN, D.; WINTERHOLER, M. An overview about debugging and verification techniques for embedded software. In: LETTNIN, D.; WINTERHOLER, M. (Ed.). *Embedded Software Verification and Debugging*. New York, NY: Springer New York, 2017. p. 1–18.

LINUX Counter Statistics. 2016. <https://www.linuxcounter.net/statistics/kernel>. Accessed: 2016-03-26.

LISBOA, E.; SILVA, L.; CHAVES, I.; LIMA, T.; BARROS, E. A design flow based on a domain specific language to concurrent development of device drivers and device controller simulation models. In: *Proceedings of th 12th International Workshop on Software and Compilers for Embedded Systems*. New York, NY, USA: ACM, 2009. (SCOPES '09), p. 53–60. ISBN 978-1-60558-696-0. Disponível em: <http://dl.acm.org/citation.cfm?id=1543820.1543830>.

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*. [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*. [S.l.: s.n.], 2011. p. 72–77.

MALER, O.; NICKOVIC, D. Formal techniques, modelling and analysis of timed and fault-tolerant systems: Joint international conferences on formal modeling and analysis of timed systmes, formats 2004, and formal techniques in real-time and fault -tolerant systems, ftrtft 2004, grenoble, france, september 22-24, 2004. proceedings. In: _____. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004. cap. Monitoring Temporal Properties of Continuous Signals, p. 152–166. ISBN 978-3-540-30206-3. Disponível em: <http://dx.doi.org/10.1007/978-3-540-30206-3_12>.

MOORE, G. E. Cramming more components onto integrated circuits. In: _____. [S.l.: s.n.], 1965. p. 114–117.

MOURA, L. D.; BJØRNER, N. Z3: An efficient smt solver. In: *Tools and Algorithms for the Construction and Analysis of Systems*. [S.l.]: Springer, 2008. p. 337–340.

NUZZO, P.; FINN, J. B.; IANNOPOLLO, A.; BAJAJ, N.; SANGIOVANNI-VINCENTELLI, A. Contract-based design of control protocols for safety-critical cyber-physical systems. November 2013.

ØHRSTRØM, P.; HASLE, P. F. *Temporal logic: from ancient ideas to artificial intelligence*. [S.l.]: Springer Science & Business Media, 1995. v. 57.

OSTRAND, T. J.; WEYUKER, E. J. The distribution of faults in a large industrial software system. In: ACM. *ACM SIGSOFT Software Engineering Notes*. [S.l.], 2002. v. 27, n. 4, p. 55–64.

PELLIZZONI, R.; MEREDITH, P.; CACCAMO, M.; ROSU, G. Hardware runtime monitoring for dependable cots-based real-time embedded systems. In: *Real-Time Systems Symposium, 2008*. [S.l.: s.n.], 2008. p. 481–491. ISSN 1052-8725.

PIKE, L.; NILLER, S.; WEGMANN, N. Runtime verification for ultra-critical systems. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, v. 7186 LNCS, n. Rv, p. 310–324, 2012. ISSN 03029743.

PRETSCHNER, A.; BROY, M.; KRUGER, I. H.; STAUNER, T. Software engineering for automotive systems: A roadmap. In: *Future of Software Engineering, 2007. FOSE '07*. [S.l.: s.n.], 2007. p. 55–71.

REINBACHER, T.; BRAUER, J.; HORAUER, M.; STEININGER, A.; KOWALEWSKI, S. Runtime verification of microcontroller binary code. *Science of Computer Programming*, v. 80, Part A, n. 0, p. 109 – 129, 2012. ISSN 0167-6423. Special section on foundations of coordination languages and software architectures (selected papers from FOCLASA'10), Special section - Brazilian Symposium on Programming Languages (SBLP 2010) and Special section on formal methods for industrial critical systems (Selected papers from FMICS'11). Disponível em: <http://www.sciencedirect.com/science/article/pii/S0167642312002109>.

RENZELMANN, M. J.; KADAV, A.; SWIFT, M. M. SymDrive : Testing Drivers without Devices. *Osdi'12*, p. 279–292, 2012.

REVEILLERE, L.; MERILLON, F.; CONSEL, C.; MARLET, R.; MULLER, G. A DSL approach to improve productivity and safety in device drivers development. *Proceedings ASE 2000. Fifteenth IEEE International Conference on Automated Software Engineering*, p. 101–109, 2000. ISSN 1938-4300. Disponível em: <http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=873655>.

RUF, J.; HOFFMANN, D.; KROPF, T.; ROSENSTIEL, W. Simulation-guided property checking based on a multi-valued ar-automata. In: *Proceedings of the Conference on Design, Automation and Test in Europe*. Piscataway, NJ, USA: IEEE Press, 2001. (DATE '01), p. 742–748. ISBN 0-7695-0993-2. Disponível em: <http://dl.acm.org/citation.cfm?id=367072.367936>.

RYZHYK, L.; CHUBB, P.; KUZ, I.; SUEUR, E. L.; HEISER, G. Automatic device driver synthesis with termite. In: *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles*. New York, NY, USA: ACM, 2009. (SOSP '09), p. 73–86. ISBN 978-1-60558-752-3. Disponível em: <http://doi.acm.org/10.1145/1629575.1629583>.

SANGIOVANNI-VINCENTELLI, A. Defining platform-based design. EE Times, 2002.

SANGIOVANNI-VINCENTELLI, A.; MARTIN, G. Platform-based design and software design methodology for embedded systems. IEEE, 2001.

SANGIOVANNI-VINCENTELLI, A. L.; DAMM, W.; PASSERONE, R. Taming dr. frankenstein: Contract-based design for cyber-physical systems. *Eur. J. Control*, v. 18, n. 3, p. 217–238, 2012. Disponível em: <http://dx.doi.org/10.3166/ejc.18.217-238>.

SARMA, S.; DUTT, N.; GUPTA, P.; NICOLAU, A.; VENKATASUBRAMANIAN, N. On-chip self-awareness using cyberphysical-systems-on-chip (cpsoc). In: *Proceedings of the 2014 International Conference on Hardware/Software Codesign and System Synthesis*. New York, NY, USA: ACM, 2014. (CODES '14), p. 22:1–22:3. ISBN 978-1-4503-3051-0. Disponível em: <http://doi.acm.org/10.1145/2656075.2661648>.

SARMA, S.; DUTT, N.; GUPTA, P.; VENKATASUBRAMANIAN, N.; NICOLAU, A. Cyberphysical-system-on-chip (cpsoc): A self-aware mpsoc paradigm with cross-layer virtual sensing and actuation. In: *Proceedings of the 2015 Design, Automation & Test in Europe Conference & Exhibition*. San Jose, CA, USA: EDA Consortium, 2015. (DATE '15), p. 625–628. ISBN 978-3-9815370-4-8. Disponível em: <http://dl.acm.org/citation.cfm?id=2755753.2755895>.

SILVA, D.; BOLZANI, L.; VARGAS, F. An intellectual property core to detect task schedulling-related faults in rtos-based embedded systems. In: IEEE. *On-Line Testing Symposium (IOLTS), 2011 IEEE 17th International*. [S.l.], 2011. p. 19–24.

SOLÍS, C.; WANG, X. A study of the characteristics of behaviour driven development. In: IEEE. *Software Engineering and Advanced Applications (SEAA), 2011 37th EUROMICRO Conference on*. [S.l.], 2011. p. 383–387.

SWIFT, M. M.; MARTIN, S.; LEVY, H. M.; EGGERS, S. J. Nooks: An architecture for reliable device drivers. In: *Proceedings of the 10th Workshop on ACM SIGOPS European Workshop*. New York, NY, USA: ACM, 2002. (EW 10), p. 102–107. Disponível em: <http://doi.acm.org/10.1145/1133373.1133393>.

TANENBAUM, A. S.; HERDER, J. N.; BOS, H. Can we make operating systems reliable and secure? *Computer*, IEEE, n. 5, p. 44–51, 2006.

TILLER, M. *Introduction to physical modeling with Modelica*. [S.l.]: Springer Science & Business Media, 2012. v. 615.

VILLARRAGA, C.; SCHMIDT, B.; BAO, B.; RAMAN, R.; BARTSCH, C.; FEHMEL, T.; STOFFEL, D.; KUNZ, W. Software in a hardware view: New models for hw-dependent software in soc verification and test. In: *Test Conference (ITC), 2014 IEEE International*. [S.l.: s.n.], 2014. p. 1–9.

WEGGERLE, A.; HIMPEL, C.; SCHMITT, T.; SCHULTHESS, P. Transaction based device driver developement. In: *MIPRO*. IEEE, 2011. p. 195–199. ISBN 978-1-4577-0996-8. Disponível em: <http://dblp.uni-trier.de/db/conf/mipro/mipro2011.html#WeggerleHSS11>.

WEISS, R. J.; RUF, J.; KROPF, T.; ROSENSTIEL, W. Efficient and customizable integration of temporal properties into systemc. In: *Applications of Specification and Design Languages for SoCs*. [S.l.]: Springer, 2006. p. 101–114.

WOLF, M. In: WOLF, M. (Ed.). *High-Performance Embedded Computing (Second Edition)*. Second edi. Boston: [s.n.], 2014. ISBN 978-0-12-410511-9.

WONGPIROMSARN, T.; TOPCU, U.; OZAY, N.; XU, H.; MURRAY, R. M. Tulip: a software toolbox for receding horizon temporal logic planning. In: *Proceedings of the 14th ACM International Conference on Hybrid Systems: Computation and Control, HSCC 2011, Chicago, IL, USA, April 12-14, 2011*. [s.n.], 2011. p. 313–314. Disponível em: <http://doi.acm.org/10.1145/1967701.1967747>.

ZHANG, Q.; ZHU, M.; CHEN, S. Automatic generation of device drivers. *ACM SIGPLAN Notices*, v. 38, n. 6, p. 60–69, 2003. ISSN 03621340. Disponível em: <http://portal.acm.org/citation.cfm?id=885638.885649>.

ZHENG, X.; JULIEN, C. Verification and validation in cyber physical systems: Research challenges and a way forward. In: *Proceedings of the First International Workshop on Software Engineering for Smart Cyber-Physical Systems*. Piscataway, NJ, USA: IEEE Press, 2015. (SEsCPS '15), p. 15–18. Disponível em: <http://dl.acm.org/citation.cfm?id=2821404.2821410>.

ZHENG, X.; JULIEN, C.; PODOROZHNY, R. M.; CASSEZ, F. Braceassertion: Runtime verification of cyber-physical systems. In: *12th IEEE International Conference on Mobile Ad Hoc and Sensor Systems, MASS 2015, Dallas, TX, USA, October 19-22, 2015*. [s.n.], 2015. p. 298–306. Disponível em: <http://dx.doi.org/10.1109/MASS.2015.15>.

ZHI-BING, W.; CHANG-YUN, L. I.; SHENG-LONG, H. U.; JUN-FENG, M. A. N. A Framework on Runtime Verification for Software Behavior. 2012.

ZIMMERMANN, H. Osi reference model–the iso model of architecture for open systems interconnection. *Communications, IEEE Transactions on*, IEEE, v. 28, n. 4, p. 425–432, 1980.

APÊNDICE A – GRAMÁTICA BNF DA TDEVC

input ::= **tdevtimport tdevcspec**
| **tdevcspec**

tdevtimport ::= **tdevtimport** import <string> ;
| import <string> ;

tdevcspec ::= device (**<string>**) { **devc_obj_def** }

devc_obj_def ::= **format_definition devc_obj_def**
| **pattern_definition** ; **devc_obj_def**
| **sto_definition devc_obj_def**
| **mem_definition** ; **devc_obj_def**
| **context_definition devc_obj_def**
| **global_state_definition**

pattern_definition ::= pattern **<string>** = **<integer>**
| pattern **<string>** = mask (**bits**)

format_definition ::= format **<string>** { **fields_decl** }

bits ::= **<mascara>**

delay_definition ::= min_delay (**<integer>** , **delay_type**)
| max_deley (**<integer>** , **delay_type**)

delay_type ::= S
| MS
| US
| NS
| PS

context_definition ::= var **<string>** ;
| var **<string>** = **<integer>** ;

sto_definition ::= internal **<string>** **reg_decl**
| external **reg_decl**

mem_definition ::= internal memory *<string>* (**<integer>** , **<integer>**)
| internal memory *<string>* (**<integer>** , **<integer>**) alias **<string>**
| external memory *<string>* (**<integer>** , **<integer>**)
| external memory *<string>* (**<integer>** , **<integer>**) alias **<string>**

reg_decl ::= register **<string>** (<integer>) { **fields_decl** } | register **<string>** (**<integer>**) alias **<string>** { **fields_decl** }
| register **<string>** (**<integer>**) = **<string>** ;
| register **<string>** (**<integer>**) alias **<integer>** = <string> ;

fields_decl ::= **field_decl:field** ;
| **field_decl:field** ; **fields_decl:fields**

field_decl ::= **rw** **<string>** [**<integer>**]
| **rw** **<string>** [**<integer>** : **<integer>**]
| reserved [**<integer>** : **<integer>**]
| reserved [**<integer>**]
| **<string>** { **fields_decl** }

rw ::= READ
| WRITE
| RW

global_state_definition ::= globalstate { **global_state_body** }

global_state_body ::= **violation_decl**
| **behavior_decl**
| **ortho_region_decl**
| **state_declaration**
| **violation_decl global_state_body**
| **behavior_decl global_state_body**
| **state_declaration global_state_body**
| **ortho_region_decl global_state_body**

state_declaration ::= **state_type** **<string>** { **state_body** }

state_type ::= state
| initialstate

state_body ::= **state_declaration**
| **violation_decl**
| **exitpoint_decl**
| **entrypoint_decl**
| **ortho_region_decl**
| **state_declaration state_body**
| **violation_decl state_body**
| **exitpoint_decl state_body**
| **entrypoint_decl state_body**
| **ortho_region_decl state_body**

ortho_region_decl ::= **orthoregion <string>** { **states_decl** }

states_decl ::= **state_declaration**
| **state_declaration states_decl**

violation_decl ::= addproperty (**violation_type_decl**) { **temporal_behavior_decl** }
| addproperty (**violation_type_decl**) <string> { **temporal_behavior_decl** }

exitpoint_decl ::= addexitpoint (**<string>**) { **reg_propositional_expr** }

entrypoint_decl ::= addentrypoint { **reg_propositional_expr** }

violation_type_decl ::= critical
| info

assign_block_decl ::= assign { **assigns_decl** } : ;
    assigns_decl ::= **assign_decl** ;
| **assign_decl** ; **assigns_decl**

assign_decl ::= **<string>** = **expr_decl**

expr_decl ::= (**expr_decl**)
| **expr_decl** + **expr_decl**
| **expr_decl** - **expr_decl**
| **expr_decl** * **expr_decl**
| **expr_decl** / **expr_decl**

| **expr_decl** « **expr_decl**
| **expr_decl** » **expr_decl**
| **<string>**
| **<integer>**

temporal_behavior_decl ::= ltlf (**ltl_logic_expr**)

propositional_logic ::= **propositional_logic**:term <-> **propositional_logic**
| **propositional_logic** -> **propositional_logic**
| **propositional_logic** **propositional_logic**
| **propositional_logic** || **propositional_logic**
| **neg_oper propositional_logic**
| (**propositional_logic**)
| **proposition_expr**

ltl_logic_expr ::= **ltl_logic_expr** U **ltl_logic_expr**
| **ltl_logic_expr** W **ltl_logic_expr**
| **ltl_logic_expr** V **ltl_logic_expr**
| [] **ltl_logic_expr**
| <> **ltl_logic_expr**
| () **ltl_logic_expr**
| **ltl_logic_expr** <-> **ltl_logic_expr**
| **ltl_logic_expr** -> **ltl_logic_expr**
| **ltl_logic_expr** **ltl_logic_expr**
| **ltl_logic_expr** || **ltl_logic_expr**
| **neg_oper ltl_logic_expr** | (**ltl_logic_expr**)
| **proposition_expr**

neg_oper ::= ~
| !
| ¬

reg$_p$*roposition_expr* ::= **reg_access**
|**reg_accesscomp_operreg_comp_expr**

proposition_expr ::= *true*
|*false*
|**<string>**
|**reg_comp_exprcomp_operreg_comp_expr**

reg_expr ::= **reg_expr** + **reg_expr**
|**reg_expr** − **reg_expr**
|**reg_expr** ∗ **reg_expr**
|**reg_expr**/**reg_expr**
|**reg_val**

reg_comp_expr ::= **reg_comp_exprcomp_operreg_comp_expr**
|**reg_expr**

reg_val ::= **reg_ref_decl**
|**<integer>**

reg_access ::= *write*(**reg_ref_decl**)
|*read*(**reg_ref_decl**)
|*write*(**any**)
|*read*(**any**)

reg_ref_decl ::= **<string>**.**reg_ref_decl**
|**<string>**

comp_oper ::===
|! =
| <=
| >=
| <
| >

APÊNDICE B – ESPECIFICAÇÕES TDEVC UTILIZADAS NOS EXPERIMENTOS

Este apêndice contém as especificações em tdevc completas dos experimentos realizados.

Especificação em TDevC da Ethernet DM9000A:

```
import ethernet_class;

device (dm9000a) {

pattern RXNOERROR = mask(....0000);

    format txStatusfmt {
        READ TJTO[7];
        READ LOSSC[6];
        READ NC[5];
        READ LATEC[4];
        READ COL[3];
        READ EC[2];
        reserved[1:0];
    }

    format multicastAddrfmt {
        RW MAB[7:0];
    }

    format physicalAddrfmt {
        RW PAB[7:0];
    }

    external register indexReg(0x00) alias INDEXREG {
        RW INDEX [15:0];
    }

    external register dataReg(0x04) alias DATAREG {
        RW DATA [15:0];
    }

    internal IntRegsProt register networkStatusReg(0x00) alias NSR {
        READ SPEED [7];
        READ LINKST [6];
        RW WAKEST [5];
        reserved[4];
        RW TX2END [3];
        RW TX1END [2];
        READ RXOV [1];
        reserved[0];
    }

    internal IntRegsProt register networkControlReg(0x01) alias NCR {
        reserved[7];
```

```
        RW WAKEEN[6];
47      reserved[5];
        RW FCOL[4];
49      READ FDX[3];
        RW LBK[2:1];
51      RW RST[0];
    }
53
    internal IntRegsProt register txControlReg(0x02) alias TCR {
55      reserved[7];
        RW TJDIS[6];
57      RW EXCECM[5];
        RW PAD_DIS2[4];
59      RW CRC_DIS2 [3];
        RW PAD_DIS1 [2];
61      RW CRC_DIS1 [1];
        RW TXREQ [0];
63  }

65  internal IntRegsProt register txStatusRegI(0x03) alias TSRI = txStatusfmt;

67  internal IntRegsProt register txStatusRegII(0x04) alias TSRII = txStatusfmt
        ;

69  internal IntRegsProt register rxControlReg(0x05) alias RCR {
        reserved[7];
71      RW WTDIS[6];
        RW DIS_LONG[5];
73      RW DIS_CRC[4];
        RW ALL[3];
75      RW RUNT[2];
        RW PRMSC [1];
77      RW RXEN[0];
    }
79
    internal IntRegsProt register rxStatusReg(0x06) alias RSR {
81      READ RF[7];
        READ MF[6];
83      READ LCS[5];
        READ RWTO[4];
85      READ PLE[3];
        READ AE[2];
87      READ CE[1];
        READ FOE[0];
89  }

91  internal IntRegsProt register receiveOverflowCounterReg(0x07) alias ROCR {
        READ RXFU[7];
93      READ ROC[6:0];
    }
95
    internal IntRegsProt register backPressureThresholdReg(0x08) alias BPTR {
97      RW BPHW[7:4];
        RW JPT[3:0];
99  }

101 internal IntRegsProt register flowControlThresholdReg(0x09) alias FCTR {
```

```
        RW HWOT[7:4];
        LWOT {
            RW AS[3];
            RW AD[2];
            RW AF[1];
            RW AG[0];
        };
        //RW LWOT[3:0];
    }

    internal IntRegsProt register rxtxFlowControlReg(0x0A) alias FCR {
        RW TXP0[7];
        RW TXPF[6];
        RW TXPEN[5];
        RW BKPA[4];
        RW BKPM[3];
        READ RXPS[2];
        READ RXPCS[1];
        RW FLCE[0];
    }

    internal IntRegsProt register multicastAddressReg7(0x1D) alias MAR7 =
        multicastAddrfmt;
    internal IntRegsProt register multicastAddressReg6(0x1C) alias MAR6 =
        multicastAddrfmt;
    internal IntRegsProt register multicastAddressReg5(0x1B) alias MAR5 =
        multicastAddrfmt;
    internal IntRegsProt register multicastAddressReg4(0x1A) alias MAR4 =
        multicastAddrfmt;
    internal IntRegsProt register multicastAddressReg3(0x19) alias MAR3 =
        multicastAddrfmt;
    internal IntRegsProt register multicastAddressReg2(0x18) alias MAR2 =
        multicastAddrfmt;
    internal IntRegsProt register multicastAddressReg1(0x17) alias MAR1 =
        multicastAddrfmt;
    internal IntRegsProt register multicastAddressReg0(0x16) alias MAR0 =
        multicastAddrfmt;

    internal IntRegsProt register wakeUpControlReg(0x0F) alias WCR {
        reserved[7:6];
        RW LINKEN[5];
        RW SAMPLEEN[4];
        RW MAGICEN[3];
        READ LINKST[2];
        READ SAMPLEST[1];
        READ MAGICST[0];
    }

    internal IntRegsProt register physicalAddressRegister5(0x15) alias PAR5 =
        physicalAddrfmt;
    internal IntRegsProt register physicalAddressRegister4(0x14) alias PAR4 =
        physicalAddrfmt;
    internal IntRegsProt register physicalAddressRegister3(0x13) alias PAR3 =
        physicalAddrfmt;
    internal IntRegsProt register physicalAddressRegister2(0x12) alias PAR2 =
        physicalAddrfmt;
    internal IntRegsProt register physicalAddressRegister1(0x11) alias PAR1 =
```

```
        physicalAddrfmt;
147     internal IntRegsProt register physicalAddressRegister0(0x10) alias PAR0 =
        physicalAddrfmt;

149
        internal IntRegsProt register generalPurposeControlReg(0x1E) alias GPCR {
151         reserved[7];
            READ GPC64[6:4];
153         RW GPC31[3:1];
            reserved[0];
155     }

157     internal IntRegsProt register generalPurposeReg(0x1F) alias GPR {
            reserved[7];
159         RW GPO[6:4];
            RW GPIO[3:1];
161         RW PHYPD[0];
        }
163
        internal IntRegsProt register txSRAMReadPointerAddress(0x22) alias TRPA {
165         READ TRPAL[15:8];
            READ TRPAH[7:0];
167     }

169     internal IntRegsProt register rxSRAMWritePointerAddressByte(0x24) alias
        RWPA {
            READ RWPAL[15:8];
171         READ RWPAH[7:0];
        }
173
        internal IntRegsProt register vendorID(0x28) alias VID {
175         READ VIDL[15:8];
            READ VIDH[7:0];
177     }

179     internal IntRegsProt register productID(0x2A) alias PID {
            READ PIDL[15:8];
181         READ PIDH[7:0];
        }
183
        internal IntRegsProt register CHIPRevision(0x2C) alias CHIPR {
185         READ CHIPR[7:0];
        }
187
        internal IntRegsProt register txControlRegII(0x2D) alias TCR2 {
189         RW LED[7];
            RW RLCP[6];
191         RW DTU[5];
            RW ONEPM[4];
193         RW IFGS[3:0];
        }
195
        internal IntRegsProt register operationTestControlReg(0x2E) alias OCR {
197         RW SCC[7:6];
            reserved[5];
199         RW SOE[4];
            RW SCS[3];
```

```
201         RW PHYOP[2:0];
        }
203
        internal IntRegsProt register specialModeControlReg(0x2F) alias SMCR {
205         RW SM_EN[7];
            reserved[6:3];
207         RW FLC[2];
            RW FB1[1];
209         RW FB0[0];
        }
211
        internal IntRegsProt register earlyTransmitControl_StatusReg(0x30) alias
            ETXCSR {
213         RW ETE[7];
            READ ETS2[6];
215         READ ETS1[5];
            reserved[4:2];
217         RW ETT[1:0];
        }
219
        internal IntRegsProt register transmitCheckSumControlReg(0x31) alias TCSCR
            {
221        reserved[7:3];
           RW UDPCSE[2];
223        RW TCPCSE[1];
           RW IPCSE[0];
225     }

227     internal IntRegsProt register receiveCheckSumControlStatusReg(0x32) alias
            RCSCSR {
            READ UDPS[7];
229         READ TCPS[6];
            READ IPS[5];
231         READ UDPP[4];
            READ TCPP[3];
233         READ IPP[2];
            RW RCSEN[1];
235         RW DCSE[0];
        }
237
        internal IntRegsProt register
            memoryDataPre_FetchReadCommandWithoutAddressIncrementReg(0xF0) alias
            MRCMDX {
239         READ MRCMDX[7:0];
        }
241
        internal IntRegsProt register memoryDataReadCommandWithAddressIncrementRegI
            (0xF1) alias MRCMDX1 {
243         READ MRCMDX1[7:0];
        }
245
        internal IntRegsProt register
            memoryDataReadCommandWithAddressIncrementRegII(0xF2) alias MRCMD {
247         READ MRCMD[7:0];
        }
249
        internal IntRegsProt register memoryDataRead_addressRegister(0xF4) alias
```

```
        MRR {
251         RW MDRAL[15:8];
            RW MDRAH[7:0];
253     }

255     internal IntRegsProt register
            memoryDataWriteCommandWithoutAddressIncrementReg(0xF6) alias MWCMDXR {
            WRITE MWCMDX[7:0];
257     }

259     internal IntRegsProt register memoryDataWriteCommandWithAddressIncrementReg
            (0xF8) alias MWCMDR {
            WRITE MWCMD[7:0];
261     }

263     internal IntRegsProt register memoryDataWrite_addressRegs(0xFA) alias MWR {
            RW MDWAL[15:8];
265         RW MDWAH[7:0];
        }
267
        internal IntRegsProt register txPacketLengthLowReg(0xFC) alias MDRALR {
269         RW MDRAL[7:0];
        }
271
        internal IntRegsProt register txPacketLengthHighReg(0xFD) alias MDRAHR {
273         RW MDRAH[7:0];
        }
275
        internal IntRegsProt register interruptStatusReg(0xFE) alias ISR {
277         READ IOMODE[7];
            reserved[6];
279         RW LNKCHG[5];
            RW UDRUN[4];
281         RW ROO[3];
            RW ROS[2];
283         RW PT[1];
            RW PR[0];
285     }

287     internal IntRegsProt register interruptMaskReg(0xFF) alias IMR {
            RW PAR[7];
289         reserved[6];
            RW LNKCHGI[5];
291         RW UDRUNI[4];
            RW ROOI[3];
293         RW ROI[2];
            RW PTI[1];
295         RW PRI[0];
        }
297
        internal IntRegsProt register EEPROM_PHYControlReg(0x0B) alias EPCR {
299         reserved[7:6];
            RW REEP[5];
301         RW WEP[4];
            RW EPOS[3];
303         RW ERPRR[2];
            RW ERPRW[1];
```

```
305         READ ERRE[0];
        }
307
        internal IntRegsProt register EEPROM_PHYAddressReg(0x0C) alias EPAR {
309         RW PHY_ADR[7:6];
            RW EROA[5:0];
311     }

313     internal IntRegsProt register EEPROM_PHYDataReg(0x0D) alias EPDR {
            RW EE_PHY_H[15:8];
315         RW EE_PHY_L[7:0];
        }
317
        internal ProtPHYRegs register BasicModeControl(0x00) alias BMCR {
319         RW RESET[15];
            RW LOOPBACK[14];
321         RW SPEEDSEL[13];
            RW AUTO_NEG_ENA[12];
323         RW POWER_DN[11];
            RW ISOLATE[10];
325         RW RESTART[9];
            RW DUPLEXMODE[8];
327         RW COLLISION_TEST[7];
            reserved[6:0];
329     }

331     mapping IntRegsProt {
            address: INDEXREG;
333         data: DATAREG;
        }
335
        mapping ProtPHYRegs {
337         address: EPAR.REGADDR;
            data: {EE_PHY_H;EE_PHY_L}
339
            addrequirement {
341            EPAR.PHYADDR = 0x01;
            }
343
            addreadrequirement{
345            EPCR = 0x0C;
            }
347
            addwriterequirement{
349            EPCR = 0x0A;
            }
351     }

353 var t1 = 1;
    var pkgcounter = 0;
355 var rxlen = 0;
    var rxlowlen = 0;
357 var txlen = 0;
    var txpkgcounter = 0;
359
    globalstate {
361
```

```
//*************************[LINK MODE]*****************************
363
    orthoregion linkState {
365     initialstate DOWN {
            addexitpoint(UP){
367             read(NSR.LINKST) == 1
            }
369     }
        state UP {
371         addexitpoint(DOWN){
                read(NSR.LINKST) == 0
373         }
        }
375 }

377 //*************************[OPER MODE]*****************************
    orthoregion ethOperationMode {
379     initialstate UNDEF_OPER_MODE {
            addexitpoint(OPER16BITS){
381             ISR.IOMODE == 0
            }
383         addexitpoint(OPER8BITS){
                ISR.IOMODE == 1
385         }
            addentrypoint{write(NCR.RST) == 1}
387     }

389     state OPER16BITS {
            addexitpoint(OPER8BITS){
391             ISR.IOMODE == 1
            }
393     }

395     state OPER8BITS {
            addexitpoint(OPER16BITS){
397             ISR.IOMODE == 0
            }
399     }
    }
401
    //*********************************************************************
403
    orthoregion PHYLayerOperation {
405     initialstate PHYDN {
            addexitpoint(PHYUP){write(GPR.PHYPD) == 0}
407     }

409     state SWRST {
            addentrypoint {write(NCR.RST) == 1}
411     }

413     state PHYUP {
            addexitpoint(PHYDN) {write(GPR.PHYPD) == 1}
415         orthoregion transmissionConfig {
                initialstate WRITEDATA {
417                 addexitpoint(SETLENGTH) {
                        write(MWCMDR)
```

```
419                         assign{txpkgcounter = txpkgcounter + 1}
                        }
421                 }
                    state SETLENGTH {
423                     addexitpoint(SETREADY) {
                            write(MDRAHR) || write(MDRALR)
425                     }
                        addexitpoint(SETLENGTH) {
427                         write(MWCMDR)
                            assign{txpkgcounter = txpkgcounter + 1}
429                     }
                    }
431                 state SETREADY {
                        addexitpoint(SETREADY) {
433                         write(MDRAHR) || write(MDRALR)
                        }
435                     addexitpoint(WRITEDATA) {
                            write(TCR.TXREQ) == 1
437                         assign{txlen = (MDRAHR << 8) + MDRALR}
                        }
439                 }
            }
441         orthoregion transmissionExec {
                initialstate WAITREADY {
443                 addexitpoint(SENDDATA){write(TCR.TXREQ) == 1}
                    //addexitpoint(SENDDATA){write(CHIPR) == RXNOERROR} TESTE
                        PARA VALIDACAO
445                 //addexitpoint(SENDDATA){write(CHIPR) == 32} TESTE PARA
                        VALIDACAO
                }
447             state SENDDATA {
                    addexitpoint(WAITREADY) {
449                         read(NSR.TX1END) == 1 || read(NSR.TX2END) == 1
                        }
451                 //addexitpoint(WAITREADY){read(NSR) == t1} TESTE PARA
                        VALIDACAO
                }
453         }
            orthoregion receive {
455         //Reception
                initialstate WAITNEWDATA {//RXPKGIDLE  {
457                 addexitpoint(READSTATUS) {
                        read(MRCMD) == 0x01
459                 }
                    addexitpoint(WAITNEWDATA) {
461                     read(MRCMD) == 0x00
                    }
463                 addexitpoint(RXPKGHEADERERR) {
                        read(MRCMDX) != 0x00 && read(MRCMDX) != 0x01
465                 }
                }
467             state READSTATUS{//RXPKGSTATUS {
                    addexitpoint(READLOWLENGTH) {
469                     read(MRCMD) == RXNOERROR
                    }
471                 }
                    state READLOWLENGTH {//RXPKGLOWLEN {
```

```
473                 addexitpoint(READHIGHLENGTH) {
                        read(MRCMD)
475                     assign{rxlowlen = MRCMD}
                    }
477             }
                state READHIGHLENGTH { // RXPKGHIGHLEN {
479                 addexitpoint(READPAYLOAD) {
                        read(MRCMD)
481                     assign{rxlen = (MRCMD << 8) + rxlowlen
                        assign{pkgcounter = 0}
483                 }
                }
485             state READPAYLOAD {//RXPKGPAYLOAD {
                    addexitpoint(READPAYLOAD) {
487                     read(MRCMD) && pkgcounter < rxlen
                        assign{pkgcounter = pkgcounter + 1}
489                 }
                    addexitpoint(READPAYLOAD) {
491                     read(MRCMDX) && pkgcounter < rxlen
                    }
493
                    addexitpoint(READSTATUS) {
495                     (read(MRCMDX) == 0x01 || read(MRCMD) == 0x01) &&
                            pkgcounter == rxlen
                    }
497
                    addexitpoint(WAITNEWDATA) {
499                     (read(MRCMDX) == 0x00 || read(MRCMD) == 0x00) &&
                            pkgcounter == rxlen
                    }
501
                    addexitpoint(RXPKGHEADERERR) {
503                     (read(MRCMD) != 0x00 && read(MRCMDX) != 0x01) &&
                            pkgcounter == rxlen
                    }
505
                }
507             state RXPKGHEADERERR {
                    addexitpoint(RXPKGHEADERERR) {
509                     read(MRCMDX) || read(MRCMD)
                    }
511                 addproperty(critical) readInError {
                        ltlf( []~(read(MRCMD)) )
513                 }
                }
515         }
            addproperty(critical) UndefinedOperMode {
517             ltlf([](~UNDEF_OPER_MODE))
            }
519
            addproperty(critical) WriteBeforeLen {
521             ltlf(
                    [](write(MDRALR) || write(MDRAHR)) -> ~WRITEDATA)
523                 )
        }
525     addproperty(critical) NeverLenAndSend {
            ltlf(
```

```
527                     []( ~(SENDDATA && SETLENGTH) )
                        )
529                 }
                addproperty(critical) TXLenPPT {
531                     ltlf(
                          [](SENDDATA -> ((OPER16BITS -> ((txlen == txpkgcounter) ||
                               (txlen == txpkgcounter-1))) && (OPER8BITS -> (txlen
                              == txpkgcounter))))
533                     )
                }
535           }
          }
537     addproperty(critical){
          ltlf( [] ~(read(any) || write(any)) )
539     }
   }
541 }
```

Especificação em TDevC do Timer:

```
device (timer_sc) {
2
    external register timerReg(0x00) alias TIMER {
4       RW TIMERF [7:0];
    }
6
    external register scalerReg(0x04) alias SCALER {
8       RW SCALERF [7:0];
    }
10
    external register DeviceControlReg(0x08) alias DCR {
12      reserved [7:1];
        RW ON [0];
14  }

16  external register DeviceStatusReg(0x0c) alias DSR {
        RW SCALERIRQ [7];
18      RW TIMERIRQ [6];
        reserved [5:0];
20  }

22  globalstate {
        orthoregion onoff{
24          initialstate OFF{
                addexitpoint(ON){write(DCR.ON) == 1}
26          }
            state ON {
28              addexitpoint(OFF){write(DCR.ON) == 0}
                addproperty(critical) noChangeAllowed {
30                  ltlf( [] (~write(TIMER) && ~write(SCALER)) )
                }
32          }
        }
34  }
}
```