



FEDERAL UNIVERSITY OF SANTA CATARINA
TECHNOLOGY CENTER
GRADUATE PROGRAM IN COMPUTER SCIENCE

Bruno Dourado Miranda

**Action and State Representations for Agent-Directed Test Generation
Targeting Multicore Designs**

Florianópolis
2026

Bruno Dourado Miranda

Action and State Representations for Agent-Directed Test Generation Targeting Multicore Designs

Thesis submitted to the Graduate Program in
Computer Science at the Federal University of
Santa Catarina in partial fulfillment of the re-
quirements for the degree of Doctor of Philo-
sophy in Computer Science.

Supervisor: Prof. Márcio Bastos Castro, Dr.

Co-supervisor: Prof. Luiz C. V. dos Santos, Dr.

Florianópolis

2026

Ficha catalográfica gerada por meio de sistema automatizado gerenciado pela BU/UFSC.
Dados inseridos pelo próprio autor.

Miranda, Bruno Dourado

Action and State Representations for Agent-Directed
Test Generation Targeting Multicore Designs / Bruno
Dourado Miranda ; orientador, Márcio Bastos Castro,
coorientador, Luiz Cláudio Villar dos Santos, 2026.

118 p.

Tese (doutorado) - Universidade Federal de Santa
Catarina, Centro Tecnológico, Programa de Pós-Graduação em
Ciência da Computação, Florianópolis, 2026.

Inclui referências.

1. Ciência da Computação. 2. Memória Compartilhada. 3.
Geração Dirigida de Testes. 4. Representação de Teste. 5.
Representação de Estado. I. Castro, Márcio Bastos. II.
Santos, Luiz Cláudio Villar dos. III. Universidade Federal
de Santa Catarina. Programa de Pós-Graduação em Ciência da
Computação. IV. Título.

Bruno Dourado Miranda

**Action and State Representations for Agent-Directed Test Generation
Targeting Multicore Designs**

O presente trabalho em nível de doutorado foi avaliado e aprovado, em 04 de dezembro de 2025, pela banca examinadora composta pelos seguintes membros:

Prof. Ismael Seidel, Dr.
Universidade Federal de Santa Catarina

Prof. Ricardo dos Santos Ferreira, Dr.
Universidade Federal de Viçosa

Prof. Rodolfo Jardim de Azevedo, Dr.
Universidade Estadual de Campinas

Certificamos que esta é a **versão original e final** do trabalho de conclusão que foi julgado adequado para obtenção do título de Doutor em Ciência da Computação.



Documento assinado digitalmente
Carina Friedrich Dorneles
Data: 03/02/2026 08:12:21-0300
CPF: ***.834.250-**
Verifique as assinaturas em <https://v.ufsc.br>

Profa. Carina Friedrich Dorneles, Dra.
Coordenadora do Programa



Documento assinado digitalmente
Marcio Bastos Castro
Data: 23/12/2025 21:43:19-0300
CPF: ***.707.720-**
Verifique as assinaturas em <https://v.ufsc.br>

Prof. Márcio Bastos Castro, Dr.
Orientador

Florianópolis, 2026.

For my family.

ACKNOWLEDGEMENTS

To Almighty God.

I would like to express my sincere gratitude to my supervisors, Márcio Castro and Luiz C. V. dos Santos. This work is the culmination of years of dedicated research and insightful discussions, and it would not have been possible without their guidance, expertise, and unwavering support through many problems.

I also extend my sincere appreciation to Gabriel G. Andrade and Marleson Graf for their valuable guidance in the study of related works and artifacts, including their own contributions.

To Luiz M. V. Pereira, Diego M. Meditsch, and Guilherme Oliveira for their collaboration and companionship throughout this academic journey. Their support and friendship have been truly valuable to me.

I am deeply grateful to my mom and dad, whose love awakened in me the desire to understand the beauty and world's complexity.

To my beloved Larissa. No words can express how important she is to me.

Finally, I thank the researchers of the Distributed Systems Research Lab (LaPeSD), Embedded Computing Lab (ECL) and the Ceos Lab for their support and collaboration.

This work was financed in part by the Conselho Nacional de Desenvolvimento Científico e Tecnológico - Brasil (CAPES) - Finance Code 001.

RESUMO

Processadores *multicore* estão no cerne da computação moderna, desde sistemas móveis e embarcados até servidores de alto desempenho e data centers. Sua eficiência depende de subsistemas complexos de memória compartilhada, regidos por *Memory Consistency Models (MCMs)* e protocolos de coerência. Garantir a correção desses subsistemas durante o projeto pré-silício é particularmente desafiador devido ao crescimento exponencial do espaço de estados com o aumento do número de núcleos e às regras de ordenação relaxadas de arquiteturas modernas como ARMv8, Power e RISC-V. A verificação funcional baseada em simulação continua sendo a metodologia predominante, na qual a geração de testes orientada por cobertura desempenha um papel crucial. No entanto, a *Random Test Generation (RTG)* convencional com restrições enfrenta limitações de eficiência sob as restrições impostas pela simulação, o que motiva o uso de técnicas de *Directed Test Generation (DTG)*. Esta tese aborda esses desafios introduzindo duas contribuições complementares que exploram *Reinforcement Learning (RL)* para aprimorar a geração de testes na verificação de memória compartilhada. Primeiramente, propomos uma representação canônica de programas de teste que codifica comportamentos fundamentais de memória compartilhada, garantindo simultaneamente completude e unicidade. Essa representação possibilita o projeto eficiente de ações para agentes de RL e evita conjuntos de testes redundantes ou restritos. Garantias teóricas e resultados experimentais demonstram que nossa abordagem supera consistentemente três geradores de última geração em termos de cobertura e detecção de erros em projetos com até 32 núcleos. Em segundo lugar, apresentamos uma representação de estado de ambiente *multicore* para geração de testes dirigida por agentes. Essa contribuição define como observar e codificar o comportamento de um *Design Under Verification (DUV)* utilizando testemunhas de execução derivados da relação *reads-from*. Ao produzir assinaturas únicas para cada execução observável e reconstruir informações de estado a partir de históricos limitados de pares ação-observação, a representação proposta aprimora a observabilidade do ambiente e permite que agentes de RL generalizem entre diferentes tarefas de verificação. Experimentos conduzidos em arquiteturas SPARC e ARMv8, utilizando os protocolos *Modified Exclusive Shared Invalid (MESI)* e *Modified Owned Exclusive Shared Invalid (MOESI)*, confirmam sua eficácia e reutilização. Ambas as contribuições foram validadas em um ambiente de verificação orientado por cobertura construído sobre o simulador *gem5*, integrado a verificadores de MCM e múltiplos analisadores de cobertura. Avaliações comparativas com geradores de última geração demonstram evolução mais rápida da cobertura e detecção antecipada de erros ao adotar as representações propostas. Em conjunto, esses resultados estabelecem novas bases para a aplicação de abordagens baseadas em aprendizado na verificação funcional de sistemas de memória compartilhada, abrindo oportunidades para uma validação pré-silício mais eficiente de arquiteturas *multicore*.

Palavras-chave: Memória Compartilhada. Geração Dirigida de Testes. Representação de Teste. Representação de Estado.

RESUMO ESTENDIDO

Introdução

Diante da crescente complexidade de requisitos computacionais, como a demanda por aceleração e paralelismo para diversas aplicações, o uso de *multicore chips* é uma realidade observada desde a computação de borda até conjuntos de servidores de alto desempenho. A maioria dos *multicore chips* de uso geral pressupõe a existência de uma memória compartilhada coerente, no entanto, o número de estados de um protocolo de coerência aumenta exponencialmente com o acréscimo de núcleos no sistema e a sua viabilidade prática é um desafio para a verificação funcional. A coerência da memória compartilhada é regida por um Modelo de Consistência de Memória (MCM), que estabelece, e implementa em *hardware*, as regras que determinam o comportamento das operações de memória durante a execução. Em arquiteturas modernas (tais como o ARMv8 e o RISC-V), as restrições de ordenamento de leituras e escritas impostas por MCM rigorosos são relaxadas, e por consequência, admitem múltiplos comportamentos válidos de um mesmo programa paralelo. Por óbvio, o aumento no espaço de comportamentos possíveis torna a validação do projeto mais complexa, visto que pequenas mudanças de projeto podem introduzir novos padrões de interação entre os núcleos, o que torna a verificação funcional um desafio significativo no projeto de *multicore chips*. Como a geração de testes *litmus* possui garantias limitadas de cobertura, a geração de testes aleatórios não-sincronizados é a alternativa preeminente até este momento para verificar a aderência de uma arquitetura a um modelo de consistência de memória compartilhada. Em cenários de verificação pré-silício, a baixa velocidade de simulação e o tempo disponível para verificação impõe uma abordagem de testes dirigida. A Figura 1 apresenta um *framework* convencional de verificação. Nesse *framework*, um *Directing Engine* é responsável por definir as diretivas de geração de testes. Essas diretivas (1) são então fornecidas ao *Test Generation Engine*, que produz o programa de teste correspondente. O programa (2) é executado em um simulador, o qual funciona como uma representação do projeto sob verificação. Durante a execução, monitores registram eventos de memória para que um verificador independente, conhecido como *checker*, possa verificar o comportamento do projeto em tempo de execução. Caso o *checker* identifique uma violação de um axioma do modelo de memória ou de qualquer outro requisito da especificação, a simulação é imediatamente interrompida e o erro é reportado pelo *framework* de verificação. Além disso, eventos de cobertura (3) são coletados ao longo da execução e, posteriormente, processados por um *Coverage Engine*, que calcula uma estimativa de cobertura. Essa estimativa (4) é então utilizada pelo *Directing Engine* para selecionar as diretivas de geração do próximo teste.

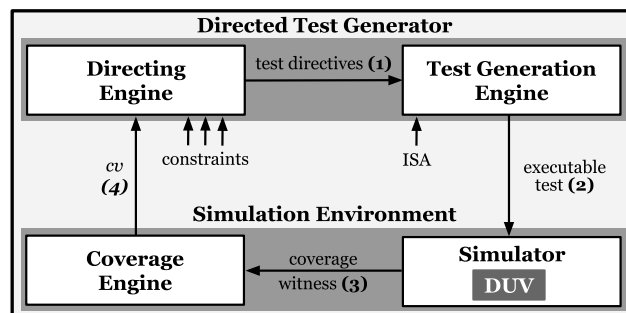


Figura 1 – *Framework* convencional de verificação

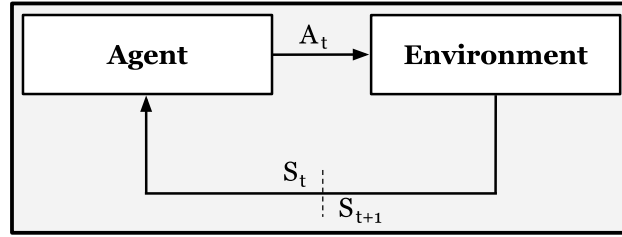


Figura 2 – Representação ilustrativa da interação entre elementos de RL

Embora os geradores dirigidos do estado da arte sejam eficazes na redução do esforço necessário para verificar a aderência do projeto ao MCM, observa-se que eles não oferecem garantias teóricas quanto à unicidade das representações nem quanto à completude do espaço de geração dos conjuntos de teste. Além disso, a reduzida observabilidade da arquitetura, limitada a simples frações de cobertura, pode restringir o uso de técnicas de Inteligência Artificial (IA). Portanto, para uma validação pré-silício mais eficiente de arquiteturas *multicore*, é desejável que o gerador seja capaz de gerar testes únicos, sem impor limitações ao espaço de geração. Além disso, ele deve ser capaz de capturar informações primárias da memória compartilhada, as quais geralmente permanecem acessíveis apenas aos *checkers*.

Objetivos

A tese tem como escopo a geração de testes dirigida para verificação funcional de memória compartilhada em *multicore chips*. Como objetivos principais, espera-se: diminuir o esforço de descoberta de erros; o aumento de cobertura acumulada em relação aos geradores reportados na literatura; e reusabilidade das técnicas propostas em diferentes protocolos de coerência, diferentes geradores dirigidos e diferentes ambientes de verificação.

Contribuições

Geradores dirigidos por dados dependem da interação com o projeto sob verificação. Portanto, para alcançar os objetivos estipulados com mínima intrusividade em algoritmos de geração, o paradigma de *Reinforcement Learning (RL)* surge como uma alternativa conveniente para o contexto de verificação de memória compartilhada. A Figura 2 ilustra a interação entre quatro componentes essenciais que compõem a técnica de RL. O agente toma uma ação A_t que estimula um comportamento do ambiente. Por sua vez, o ambiente responde ao estímulo com uma representação do estado S_{t+1} após a tomada da ação. O ciclo de interações agente-ambiente possibilita a exploração do espaço de geração e a avaliação de seus efeitos ao longo do tempo. Entretanto, um dos principais desafios para o uso de RL na otimização de um problema-alvo é a definição de uma representação adequada do estado do ambiente e das ações, de modo que seja possível ao agente aprender uma política que maximize o retorno esperado a partir de cada interação com o ambiente. Neste escopo, a tese relata duas contribuições complementares:

- Uma representação canônica de programas de teste que codifica comportamentos primais de memória compartilhada, garantindo simultaneamente completude e unicidade de representação. Isso possibilita o projeto eficiente de ações para agentes de RL e evita conjuntos de testes redundantes ou restritos;
- Uma representação de estado de ambiente *multicore* para geração de testes dirigida por agentes. Essa contribuição define como observar e codificar o comportamento de

um projeto sob verificação utilizando testemunhas de execução derivadas da relação lê-de. Ao produzir assinaturas únicas para cada execução observável e reconstruir informações de estado a partir de históricos limitados de pares ação–observação, a representação proposta aprimora a observabilidade do ambiente e permite que agentes de RL generalizem entre diferentes tarefas de verificação.

Metodologia

Para avaliação experimental das contribuições propostas foram adotados os seguintes passos metodológicos: construção de quatro geradores de teste baseados em aprendizado por reforço; uso de geradores publicados na literatura como base comparativa; uso de verificadores e de erros artificiais disponíveis na literatura; simulação de execução dos geradores para diferentes representações, com ou sem falhas de projeto; comparação de evolução de cobertura, assim como a eficácia e esforço dos geradores em expor erros sob as mesmas condições experimentais.

Resultados e Discussão

A primeira contribuição desta tese foi incorporada em um gerador baseado em Aprendizado por Reforço disponível na literatura. Além disso, as ações do novo gerador (nomeado LBG) foram construídas com base na contribuição proposta. Em comparação a dois geradores na literatura que utilizam uma representação de teste que codifica diretivas a um módulo gerador de testes pseudo-aleatório, observou-se um aumento de cobertura acumulada em diferentes protocolos e diferentes métricas para uma arquitetura de 32 núcleos. Quando comparado a um terceiro gerador (nomeado MTG), baseado em Programação Genética e que possui uma codificação de teste de propósito geral, LBG não foi capaz de atingir a mesma cobertura acumulada em apenas uma métrica de um protocolo. No entanto, as restrições impostas por MTG podem excluir a melhor solução do espaço de geração e isso pode ser evidenciado no esforço de verificação. Em diferentes episódios de verificação, LBG expôs os erros artificiais, em média, com menor cobertura e tempos inferiores (em alguns cenários em uma ordem de magnitude) aos de MTG. As garantias teóricas da contribuição proposta impedem a geração de testes redundantes e a limitação do espaço de geração de testes. Já as evidências experimentais indicam que a representação de teste proposta fornece bases adequadas para definir ações, pois codifica comportamentos primários efetivos.

A segunda contribuição desta tese foi avaliada em quatro tarefas de verificação na arquitetura ARMv8, envolvendo diferentes métricas de cobertura e diferentes quantidades de núcleos sob o protocolo 2-level MOESI. Em comparação a uma representação de estado do ambiente reportada na literatura, observou-se uma evolução de cobertura superior em todas as tarefas quando um gerador baseado em Aprendizado por Reforço utiliza a representação de estado proposta (nomeado DQG*). Notavelmente, mesmo quando as observações foram deterioradas para valores de cobertura acumulada em um gerador variante de DQG*, tal variante (nomeada DQG+) apresentou uma evolução de cobertura superior ao gerador reportado na literatura. Isso indica que a captura do histórico de interações do agente com o ambiente habilitou o aprendizado. O mesmo comportamento ocorreu quando DQG* foi comparado a um outro gerador que possui uma política independente de estado e ao MTG. Finalmente, quando comparado a DQG+, o gerador DQG* mostrou evolução de cobertura superior, indicando que a observação adotada pela representação de estado proposta é vantajosa para o problema-alvo. Em diferentes episódios de verificação analisados, DQG* obteve maior eficácia na exposição de erros e um

menor esforço de verificação. Em contraste, MTG foi incapaz de expor os erros em todos os episódios. As evidências experimentais indicam que quando observações de testemunha de execução baseadas em uma relação de ordem são codificadas como assinaturas únicas, elas tendem a maior evolução de cobertura e menor esforço de verificação ao comparadas a outras técnicas reportadas na literatura.

Considerações Finais

A tese teve como escopo a geração de testes dirigida para verificação funcional de memória compartilhada em *multicore chips*. Ela assume um gerador composto por um agente baseado em Aprendizado por Reforço, o qual interage com um ambiente que contém um projeto sob verificação. A tese apresentou duas contribuições, que em sinergia, são uma alternativa promissora às técnicas convencionais: a primeira contribuição foi uma representação de teste canônica, que fornece garantias teóricas de não-redundância e que não limita o espaço de busca, permitindo a construção de ações efetivas. Tal representação é independente de métricas e protocolos de coerência, o que a torna reutilizável em geradores baseados em diferentes técnicas de direcionamento. Quando incorporada a um gerador baseado em Aprendizado Por Reforço, evidências experimentais indicaram que a representação de teste proporciona uma exploração satisfatória do espaço de busca e permite a formulação de ações efetivas que aumentam a exploração de agentes. Em cobertura máxima, o gerador protótipo foi superior em 75% dos cenários analisados em comparação a outros geradores reportados na literatura. Em esforço de verificação, o gerador protótipo apresentou desempenho equivalente ou superior a um gerador baseado em Programação Genética ao expor dois erros artificiais reportados na literatura. A tese também apresentou uma segunda contribuição: uma representação de estado de ambiente específica ao domínio, baseada em uma relação de ordem. A representação de estado codifica observações de testemunhas de execução do ambiente de verificação em assinaturas comprovadamente únicas. Evidências experimentais mostraram que geração de testes direcionadas por agentes necessitam armazenar interações entre o agente e o ambiente, assim como depender de observações específicas ao domínio. Quando incorporada a um gerador, a representação de estado proporcionou uma evolução mais rápida de cobertura quando comparada à de quatro geradores distintos. Sob a ótica de esforço de verificação, o gerador atingiu um desempenho superior ao trabalho correlato baseado em Programação Genética.

Palavras-chave: Memória compartilhada. Geração Dirigida de Testes. Representação de teste. Representação de estado.

ABSTRACT

Multicore processors are at the core of modern computing, from mobile and embedded systems to high-performance servers and data centers. Their efficiency relies on complex shared-memory subsystems governed by Memory Consistency Models (MCMs) and coherence protocols. Ensuring the correctness of these subsystems during pre-silicon design is particularly challenging due to the exponential growth of states with the number of cores and the relaxed ordering rules of modern architectures such as ARMv8, Power, and RISC-V. Simulation-based functional verification remains the predominant methodology, where coverage-driven test generation plays a crucial role. However, conventional constrained Random Test Generation (RTG) struggles with efficiency under simulation constraints, motivating the use of Directed Test Generation (DTG) techniques. This thesis addresses these challenges by introducing two complementary contributions that leverage Reinforcement Learning (RL) to enhance test generation for shared-memory verification. First, we propose a canonical test-program representation that encodes fundamental shared-memory behaviors while ensuring both completeness and uniqueness. This representation enables effective action design for RL agents and avoids redundant or restricted test suites. Theoretical guarantees and experimental results demonstrate that our approach consistently outperforms three state-of-the-art generators in terms of coverage and error detection across 32-core designs. Second, we present a multicore environment state representation for agent-directed test generation. This contribution defines how to observe and encode the behavior of a Design Under Verification (DUV) using execution witnesses derived from the *reads-from* relation. By producing unique signatures for every observable execution and reconstructing state information from bounded histories of action-observation pairs, the proposed representation improves environment observability and enables RL agents to generalize across verification tasks. Experiments on SPARC and ARMv8 architectures, using Modified Exclusive Shared Invalid (MESI) and Modified Owned Exclusive Shared Invalid (MOESI) protocols, confirm its effectiveness and reusability. Both contributions were validated within a coverage-driven verification framework built on the gem5 simulator, integrated with MCM checkers and multiple coverage analyzers. Comparative evaluations against state-of-the-art generators demonstrate faster coverage evolution and earlier error diagnosis when adopting the proposed representations. Together, these results establish new foundations for applying learning-based approaches to functional verification of shared-memory systems, opening opportunities for more efficient pre-silicon validation of multicore architectures.

Keywords: Shared Memory. Directed Test Generation. Test Representation. State Representation.

LIST OF FIGURES

Figura 1 – <i>Framework</i> convencional de verificação	13
Figura 2 – Representação ilustrativa da interação entre elementos de RL	14
Figure 3 – Generic structure of a multicore chip.	34
Figure 4 – A state-of-the art coverage-driven framework.	36
Figure 5 – MSI protocol	43
Figure 6 – MOESI protocol	44
Figure 7 – Illustration of valid outcomes from a parallel program	45
Figure 8 – The agent–environment interaction in an MDP.	47
Figure 9 – A typical verification framework when using a test generator and an external checker	63
Figure 10 – Synthesis of a test program from an encoding $e = \{e_1, e_2\}$	71
Figure 11 – A verification framework for an encoding-based DTG approach	73
Figure 12 – Anatomy of the Directing Engine for the proposed approach	73
Figure 13 – Anatomy of the Test Generation Engine for the proposed approach	73
Figure 14 – Impact of distinct sets of actions on coverage evolution	79
Figure 15 – Median <i>structural</i> coverage evolution for 32-core designs as a function of the required time	82
Figure 16 – Median <i>functional</i> coverage evolution for 32-core designs as a function of the required time	82
Figure 17 – Discovery constellations for 2-level MOESI design errors	85
Figure 18 – Discovery constellations for 3-level MESI design errors	86
Figure 19 – A reinforcement learning approach to DTG	93
Figure 20 – An observation captured by a joint witness to execution instances $I =$ $\{<, < \}$, and its representation as a signature.	95
Figure 21 – Median (cumulative) coverage evolution for 24-hour verification episodes	99
Figure 22 – Coverage-time error discovery constellations for faulty designs	100

LIST OF TABLES

Table 1 – Illustrative example of an incoherent multicore architecture	41
Table 2 – The relaxed order allowed for distinct memory models	45
Table 3 – Constraint exploitation for distinct RTG approaches.	52
Table 4 – Adopted acronyms for existing DTG techniques.	53
Table 5 – Model-based, data-driven, and hybrid approaches to DTG.	53
Table 6 – Meaning of the symbols used to define the canonical representation . . .	69
Table 7 – Median structural coverage reached for a given time limit	80
Table 8 – Median functional coverage reached for a given time limit	81
Table 9 – Median time for error diagnosis (in seconds)	83
Table 10 – Generators used to evaluate our state representation	97
Table 11 – Median error discovery times (in seconds)	101

LIST OF ALGORITHMS

1	ENCODER(g, p, a)	75
2	ENCODE(g, p)	76
3	RECODE(e, g, p, a)	76
4	DIRECTING ENGINE($n_{\min}, n_{\max}, s_{\min}, s_{\max}, p$)	77
5	DQG-LEARNING-RATE-UPDATE(i, η_{i-1}, η_{th})	115
6	DQG-BATCH-SIZE-UPDATE(i, bs_{i-1}, bs_{th})	115
7	DQG-NEURONS-UPDATE(i, nn_{i-1}, nn_{th})	115
8	SUITABLE-PARAMETERS-DQN($\eta_{th}, bs_{th}, nn_{th}$)	116

LIST OF ABBREVIATIONS AND ACRONYMS

BNG	Bayesian Network Generator.	54
CSP	Constraint Satisfaction Problem.	52
DFSM	Dichotomic Finite State Machine.	54
DNN	Deep Neural Network.	48
DQG	Deep-Q Generator.	38
DQN	Deep-Q Network.	38
DTG	Directed Test Generation.	35
DUV	Design Under Verification.	34
DV	Data Value.	42
EDA	Electronic Design Automation.	104
ETG	Euler Tour Generator.	53
FSM	Finite State Machine.	34
GP	Genetic Programming.	54
HTG	Hybrid Test Generator.	38
ISA	Instruction Set Architecture.	38
LBG	Learning-Based Generator.	38
LRU	Least Recently Used.	67
MCM	Memory Consistency Model.	33
MDP	Markov Decision Process.	47
MESI	Modified Exclusive Shared Invalid.	34
ML	Machine Learning.	35
MOESI	Modified Owned Exclusive Shared Invalid.	34
MOSI	Modified Owned Shared Invalid.	34
MRSB	Multiple Relaxed Scoreboard.	38
MSI	Modified Shared Invalid.	42
MTG	McVersi Test Generator.	38
PSO	Partial Store Order.	44

QSG	Quotient Space Generator.	53
RAG	Random Agent Generator.	38
RC	Relaxed Consistency.	44
RL	Reinforcement Learning.	37
RLG	Reinforcement Learning Generator.	38
RTG	Random Test Generation.	35
SC	Sequential Consistency.	44
SCG	Steep Coverage Generator.	54
SWMR	Single-Writer, Multiple Reader.	42
TSO	Total Store Ordering.	38
WO	Weak Ordering.	44

LIST OF SYMBOLS

W	Write operation
R	Read operation
L	Load operation
S	Store operation
e	Encoding
cv	Coverage value
C	Cumulative coverage
P	Test program
$\mathbb{G}$	Generation space
$\mathbb{E}$	Encoding space
o_j^a	J-th memory operation issued by thread a
S	State
R	Reward
A	Action
O	Observation
h	Action-observation history bound
ρ	Reward conversion function
τ	State conversion function
σ	Observation conversion function
sig	Signature function
EW	Execution witness
$<$	Execution instance
θ	Upper bound for the number of stores
n	Number of memory operations
s	Number of shared locations
p	Number of threads
k	Number of cache indices
m	Number of operation types
E	Set of encodings
R	Set of rewards
I	Set of execution instances
$\mathcal{L}$	Set of load operations
S	Set of states
$\mathcal{A}$	Set of actions
$\mathcal{O}$	Set of observations

CONTENTS

1	INTRODUCTION	33
1.1	SHARED-MEMORY BEHAVIOR IN MULTICORE SYSTEMS . . .	33
1.2	DIRECTED TEST GENERATION	34
1.3	CONTRIBUTIONS	36
1.4	COVERAGE-DRIVEN FRAMEWORK AND SOFTWARE ARTIFACTS	37
1.5	ORGANIZATION OF THE THESIS	39
2	FUNDAMENTAL CONCEPTS	41
2.1	CACHE COHERENCE	41
2.1.1	Coherence Protocols	42
2.1.2	Consistency Rules	44
2.1.3	Coverage Metrics	45
2.2	MACHINE LEARNING (ML)	46
2.2.1	Reinforcement Learning (RL)	47
2.2.2	Deep Q-Learning	48
3	RELATED WORK	51
3.1	RANDOM TEST GENERATION (RTG)	51
3.2	DIRECTED TEST GENERATION (DTG)	52
3.2.1	Model-based Generators	53
3.2.2	Data-driven Generators	54
3.2.3	Hybrid Generation	55
3.3	DISCUSSION	55
3.3.1	Limited DUV Independence	56
3.3.2	Limited Exploration of the Test Generation Space	56
3.3.3	Limited Exploitation of Constraints	57
3.3.4	The Lack of a Proper Test Representation	57
3.3.5	The Lack of a Proper DUV Observation	58
4	A CANONICAL TEST REPRESENTATION FOR VERIFI- CATION OF SHARED-MEMORY BEHAVIOR IN MULTI- PROCESSOR SYSTEMS	61
4.1	INTRODUCTION	61
4.1.1	Verification requirements and goals	61
4.1.2	Main approaches to test generation	62
4.1.3	Methodology and high-level ideas	63
4.1.4	Contribution	64
4.2	RELATED WORK	65

4.3	THE PROPOSED TEST REPRESENTATION	66
4.3.1	Pragmatic Requirements	67
4.3.1.1	<i>Deliberate Lack of Synchronization</i>	67
4.3.1.2	<i>Ability to Induce Permission Requests</i>	67
4.3.1.3	<i>Ability to Induce Eviction Requests</i>	67
4.3.2	Theoretical Requirements	68
4.3.3	The Canonical Encoding	68
4.3.4	Theoretical Guarantees	69
4.3.5	Implementation and Example	70
4.4	TEST GENERATION AS AN OPTIMIZATION PROBLEM	71
4.5	THE LEARNING-BASED GENERATOR	72
4.5.1	Design of Actions based on the Canonical Representation . .	74
4.5.2	Encoder	75
4.5.3	Directing Engine	76
4.6	EXPERIMENTAL EVALUATION	77
4.6.1	Infrastructure and Setup	78
4.6.2	Action Effectiveness under the Proposed Approach	79
4.6.3	Impact on Coverage for a Given Time Limit	80
4.6.4	Coverage Evolution with Time	81
4.6.5	Impact on Design Error Diagnosis	83
4.7	CONCLUSIONS AND PERSPECTIVES	87
5	MULTICORE ENVIRONMENT STATE REPRESENTATION FOR AGENT-DIRECTED TEST GENERATION	89
5.1	INTRODUCTION	89
5.2	MOTIVATION IN FACE OF RELATED WORK	90
5.3	THE PROPOSED APPROACH	91
5.3.1	Background in Brief	91
5.3.2	Agent-based Approach to DTG	92
5.4	THE PROPOSED STATE REPRESENTATION	93
5.4.1	Choice of suitable observation and proper representation . .	93
5.4.2	Illustrative example	95
5.4.3	Recovering of state information from observations	96
5.5	EXPERIMENTAL EVALUATION	96
5.5.1	Coverage Evolution with Time	98
5.5.2	Impact on Error Discovery	98
5.6	CONCLUSIONS	101
6	CONCLUSIONS AND PERSPECTIVES	103
6.1	FUTURE WORK	104

BIBLIOGRAPHY	107
------------------------	-----

APPENDIX A – METHODOLOGY TO SELECT THE HYPER- PARAMETERS OF THE NEURAL NET- WORK	113
--	-----

A.1	METHODOLOGY GOAL	113
A.2	ARBITRARY CHOICES	113
A.3	HYPER-PARAMETER ϵ	113
A.4	STRIPING DOWN	113
A.4.1	The Q-Network	113
A.4.2	The target problem	114
A.5	SUITABLE HYPER-PARAMETER VALUES	114
A.5.1	Guidelines for validating a new value's impact on learning . .	114
A.5.1.1	<i>Criterion for hyper-parameter update</i>	<i>114</i>
A.5.2	Step Zero - threshold values	114
A.5.3	Tuning routines	115
A.6	HYPER-PARAMETER TUNING	116
A.6.1	For a given task	116
A.6.2	For the task set	116

1 INTRODUCTION

Multicore processors have become the cornerstone of modern computing, powering devices across the entire spectrum of applications, from energy-constrained mobile phones and embedded systems to high-performance servers, and data centers. Their widespread adoption is driven by the fundamental limitations of single-core performance scaling, where increases in clock frequency are no longer feasible due to power and thermal constraints. By integrating multiple cores on a single chip, designers can exploit parallelism to improve throughput, energy efficiency, and responsiveness, while meeting the diverse requirements of consumer, enterprise, and scientific workloads (Hennessy; Patterson, 2017). As a result, multicore architectures are not only pervasive in everyday personal devices but also indispensable in the backbone of Cloud Computing services.

Commercial multicore chips are available for architectures such as x86-64, ARMv8, and Power. RISC-V multicore chips can be expected in the near future due to the convenience of the open architecture (Patterson, 2018; Greengard, 2020; Laasch, 2025). Therefore, the need for new multicore chip designs goes beyond the demand of major server vendors because designs have to comply with the distinct requirements of different computing classes. Such requirements define not only the proper number of cores, but also the adequate memory hierarchy.

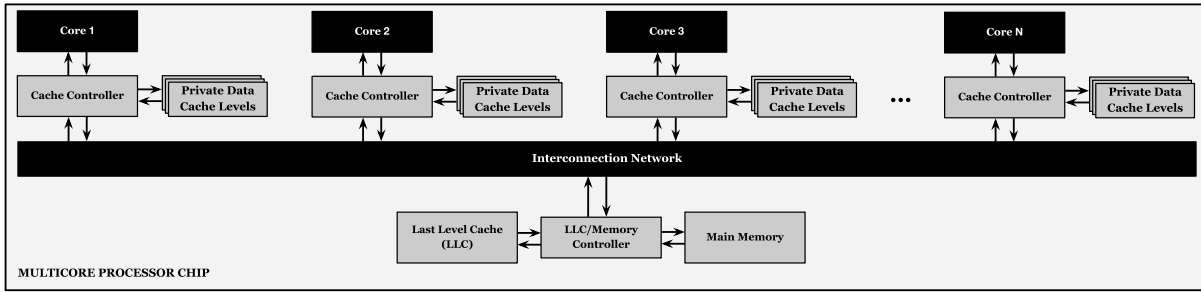
Shared memory is the predominant communication mechanism in multicore chips, supported by a complex hardware infrastructure that includes queues, buffers, multiple levels of private caches, and interacting cache controllers across core domains. As multicore architectures continue to grow in complexity, verifying the correctness of shared-memory subsystems has become increasingly challenging. These subsystems are governed by Memory Consistency Models (MCMs) (Gharachorloo et al., 1990; Nagarajan et al., 2020), which specify the set of valid behaviors for concurrent programs. Ensuring compliance with MCMs is particularly difficult because a single concurrent test program may yield multiple valid outcomes. As a result, the design of such subsystems is challenging, error-prone, and requires sophisticated verification techniques.

Given the pervasive adoption of multicore chips across diverse computing domains, the critical importance of their design validation, and the inherent complexity of the hardware mechanisms that implement shared-memory behavior, this thesis focuses on the *functional verification* of shared-memory behavior in the design phase of multicore architectures.

1.1 SHARED-MEMORY BEHAVIOR IN MULTICORE SYSTEMS

Figure 3 illustrates a generic structure for a multicore chip. All cores can execute load and store operations that share the same address space. Caches are used for hiding the latency to the main memory. Each core has one or more levels of private caches and

Figure 3 – Generic structure of a multicore chip.



Source: Adapted from Nagarajan et al. (2020).

a shared last-level cache.

The allowed *behavior* of load and store operations sharing the same address space and belonging to the multiple threads of a concurrent program is specified by an MCM, which essentially defines whether the program order between pairs of memory operations can be relaxed or not, and to which extent stores may appear as atomic operations from the point of view of all cores. Early MCMs, such as Sequential Consistency (Lamport, 1978), fully preserved program order and store atomicity. However, contemporary architectures such as ARMv8, Power, and RISC-V rely on largely relaxed MCMs such as Weak Ordering or Release Consistency for higher performance (Hennessy; Patterson, 2017). Being less constrained, these MCMs allow a much larger number of valid behaviors than Sequential Consistency, making it more difficult to verify if the hardware structure properly implements them.

Although caches help mitigate main memory latency, their use introduces challenges in maintaining data coherence, as multiple cores may store local copies of the same shared memory *location* in their private caches. This situation is harmless as long as all copies are read-only. Problems arise, however, when one core updates its local copy: if other cores are unable to observe the new value, the system falls into incoherence. Coherence protocols such as Modified Exclusive Shared Invalid (MESI), Modified Owned Shared Invalid (MOSI), and Modified Owned Exclusive Shared Invalid (MOESI) (Hennessy; Patterson, 2017) are used to prevent incoherence. A protocol defines *states* and *transitions* for the Finite State Machine (FSM) of each cache controller. The overall protocol space corresponds to the product of the FSMs of each core domain. Since the states of the product FSM grow exponentially with the number of cores, coherent shared memory verification is challenging, because proper coverage of states or transitions has to be obtained without explicit enumeration.

1.2 DIRECTED TEST GENERATION

Simulation-based design verification remains the predominant methodology for validating a target multicore Design Under Verification (DUV). In this approach, progress

is typically measured through a metric known as *coverage*, which quantifies the fraction of the design structure or behavior exercised during verification. To complement coverage analysis, verification also requires mechanisms capable of determining whether the observed outcomes conform to those expected from a golden model. Consequently, accurate error diagnosis becomes a fundamental requirement. The ultimate goal of simulation-based verification is to ensure that no design errors are exposed before either full coverage is attained or a predefined time limit is reached. Achieving this goal, however, hinges on the ability to generate effective test programs that not only accelerate error detection but also maximize coverage.

When applied to shared memory subsystems, these general requirements encounter unique challenges. Unlike other domains where input stimuli can be directly synthesized, the verification of multicore designs typically relies on the generation of concurrent *test programs*, which indirectly induce stimuli during execution. Since the underlying MCM allows different interleavings of memory operations, a single program can lead to multiple valid executions. This variability precludes the existence of a simple one-to-one golden model for expected behavior, as is common in other verification domains. At first glance, one might attempt to address this difficulty by restricting test generation to synchronized programs. However, synchronization primitives, while preventing data races, also suppress collisions between memory operations from different threads, and such collisions are essential for both increasing coverage and exposing subtle errors.

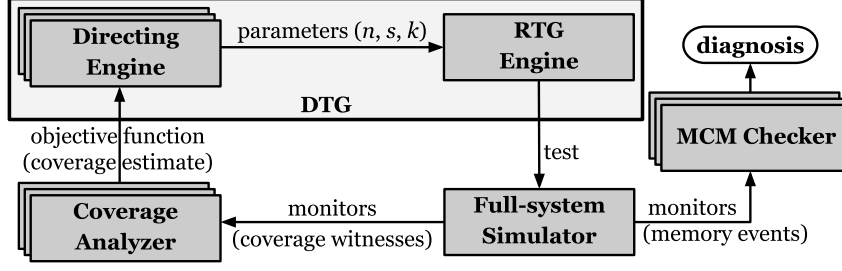
Constrained Random Test Generation (RTG) has long been the conventional approach used for design verification to avoid the enumeration of the interactions between processors through shared memory (Andrade; Graf; Santos, 2016; Andrade; Graf; dos Santos, 2019). By generating each thread independently at random, under range constraints, the likelihood of covering diverse interactions increases with thread length, making RTG particularly efficient for multicore chip testing where large test programs can run directly on hardware prototypes. In the context of design verification, however, programs must be much shorter due to the slower execution of simulations, which limits the effectiveness of RTG.

To attain adequate coverage with shorter test programs, Directed Test Generation (DTG) is generally favored (Adir et al., 2004; Andrade et al., 2018; Pfeifer et al., 2020; Andrade et al., 2020). DTG approaches often rely on heuristics or, more recently, Machine Learning (ML) techniques to explore the protocol state space more effectively, aiming to maximize coverage while minimizing verification time. Most existing DTG methods leverage an external MCM checker, which ensures that the axioms of the target memory consistency model are satisfied by each execution instance of a concurrent program.¹

Due to such RTG legacy, several DTG approaches rely on an RTG engine that is

¹ In the context of shared-memory *testing*, an MCM checker usually operates in a postmortem fashion, analyzing execution traces after a program has completed. In contrast, in design *verification*, an MCM checker is a runtime component that halts simulation immediately upon detecting a violation.

Figure 4 – A state-of-the art coverage-driven framework.



Source: Andrade et al. (2018).

driven by a *Directing Engine* over the pre-silicon verification phase. To improve coverage evolution, the latter dynamically modifies the *parameters*, serving as constraints to the former. Figure 4 shows a framework proposed by Andrade et al. (2018) that employs a RTG-driven approach to DTG. It consists of a Directing Engine coupled with an RTG Engine. Each generated test is executed on a simulated DUV. Monitors are inserted in appropriate points of the design representation, and they capture memory events, which are analyzed by an MCM checker. If a monitored event happens to violate the MCM, the checker stops the simulation immediately and indicates the error. Besides, other monitors are inserted to capture coverage witnesses, which are handled by a *Coverage Analyzer* that provides a cumulative coverage estimate for the Directing Engine. Depending on coverage evolution, the Directing Engine decides convenient new values for the RTG parameters to induce a next test that is likely to improve coverage. As far as no error is found, several tests are generated and executed until proper coverage is reached. In that approach, three-generation parameters are dynamically modified by the Directing Engine: number of memory operations (n), number of shared locations (s), and number of locations mapping to the same cache line (k).

1.3 CONTRIBUTIONS

This thesis addresses the above challenges by combining two complementary contributions, each documented in a different published paper:

- **A canonical test representation for verification of shared-memory behavior in multiprocessor systems (Miranda; Castro; Santos, 2025).** We introduce a canonical representation for test programs that encodes fundamental shared-memory behaviors to be exercised at runtime, providing a foundation for the development of new test generators. The proposed representation guarantees both completeness and uniqueness by inducing equivalence classes that can be fully and unambiguously encoded, thereby ensuring that the search space is not restricted. Beyond its theoretical properties, we demonstrate through experiments that the

representation is well suited to learning-based test generation, as it enables the design of effective actions for Reinforcement Learning (RL) agents. To validate this claim, we implemented a generator guided by a minimalist RL agent, whose actions were defined using our encoding, and compared it against three state-of-the-art generators on 32-core designs. Within a fixed time budget, our generator consistently achieved higher coverage and faster error diagnosis in three out of four verification scenarios. These results, combined with the theoretical guarantees, show that our representation provides a solid basis for designing efficient actions while avoiding both redundant test generation and unnecessary limitations in test suites.

- **A multicore environment state representation for agent-directed test generation (Miranda et al., 2025).** We introduce a novel RL approach to DTG that emphasizes the construction of suitable state representations for environments containing a DUV in multicore systems. We address three key aspects: what should be observed, how these observations should be represented, and how they can be leveraged to recover environment state information. To this end, we introduce the notion of a witness to execution instances of concurrent programs as a principled means of capturing the response of a multicore environment to an agent’s actions. This concept repurposes the well-known *reads-from relation*, which is traditionally used in the synthesis of litmus tests, into a state representation for RL, an application not previously explored. We further show that test generation can be constrained to guarantee a unique and provably unambiguous representation for every observable execution witness induced by a test suite. Finally, we demonstrate how environment state information can be effectively reconstructed from a bounded history of action-observation pairs, enabling reinforcement learning agents to reason more efficiently about verification tasks.

1.4 COVERAGE-DRIVEN FRAMEWORK AND SOFTWARE ARTIFACTS

Since 2013, the Embedded Computing Lab (ECL) has developed a number of software artifacts for shared-memory verification. For the purposes of this thesis, the corresponding source codes and related data were made available in a private repository. Several of these artifacts were directly reused to build a verification framework (Figure 4), which was employed to implement and conduct the experiments to validate the contributions of this thesis. A brief description of these software artifacts is provided below:

1. **State-of-the-art generators:** The verification framework shown in Figure 4 incorporates software artifacts that were reused to implement different DTGs described in the literature. These DTGs were employed to perform experiments enabling a direct comparison with the contributions of this thesis under identical conditions, which may differ from those reported in the original publications. Specifically, we

considered the McVersi Test Generator (MTG) (Elver; Nagarajan, 2016), Reinforcement Learning Generator (RLG) (Pfeifer et al., 2020), and Hybrid Test Generator (HTG) (Andrade et al., 2020) as baselines for comparison. All of these generators share the same framework modules, differing only in their Directing Engines. With the exception of the MTG, they reuse the RTG engine introduced by Andrade, Graf & dos Santos (2019).

2. **New DTG approaches:** We designed two learning-based generators that employ RL agents. The Learning-Based Generator (LBG) is a variant of RLG, featuring a distinct set of actions and a novel test representation. The Deep-Q Generator (DQG) builds on the Deep-Q Network (DQN) approach (Mnih et al., 2013), reusing the LBG action set and test representation, but introducing a new environment state representation. To further isolate and evaluate the effect of state representation, we also developed a Random Agent Generator (RAG). Since the RAG’s policy is state-independent, it serves as a baseline for measuring the impact of this contribution in relation to prior work.
3. **Full-system simulator:** The verification framework shown in Figure 4 builds on the public-domain gem5 simulation environment (Binkert et al., 2011).² The environment was customized as an executable design representation, with monitors inserted at key points to observe relevant aspects of the design. The gem5 framework was selected for its broad support of distinct Instruction Set Architecture (ISA) targets and, in particular, its rich variety of coherence protocols. Our experiments focused on SPARC and ARMv8 architectures, employing 3-level MESI and 2-level MOESI protocols. To rigorously evaluate our contributions against related work, faulty variants of each design representation were created by introducing a single error into each configuration.
4. **Coverage analyzers:** A dedicated instance of the coverage analyzer is provided for each coverage metric employed. Specifically, instances were implemented to support both *structural* (Elver; Nagarajan, 2016) and *functional* (Andrade et al., 2020) coverage metrics, for each of the coherence protocols described above.
5. **MCM compliance:** Two MCM checkers were employed to verify the shared-memory behavior of each simulation. The Multiple Relaxed Scoreboard (MRSB) (Freitas; Rambo; Santos, 2013) enforces strict store atomicity at the implementation level, thus targeting conservative designs and the Total Store Ordering (TSO) memory model adopted by SPARC. In contrast, EveCheck (Graf; Andrade; Santos, 2023) supports relaxed store atomicity at the implementation level, accommodating both conservative and aggressive designs, and targets the ARMv8 memory model.

² Available at <https://gem5.org>

1.5 ORGANIZATION OF THE THESIS

The thesis is organized as follows:

- **Chapter 2** introduces the fundamental concepts required to understand the main problems addressed in this thesis;
- **Chapter 3** surveys related work and highlights how the contributions of this thesis advance beyond prior approaches;
- **Chapter 4** presents the first contribution of the thesis: a canonical test representation for the verification of shared-memory behavior in multiprocessor systems. It reproduces the article published in *ACM Transactions on Design Automation of Electronic Systems* (TODAES) (Miranda; Castro; Santos, 2025);
- **Chapter 5** presents the second contribution of the thesis: a multicore environment state representation for agent-directed test generation. It reproduces the paper published in the proceedings of the *ACM/IEEE Design Automation Conference* (DAC 2025) (Miranda et al., 2025);
- **Chapter 6** summarizes the main conclusions and outlines directions for future research.

2 FUNDAMENTAL CONCEPTS

This chapter reviews the essential concepts required to address the target problem of this thesis: generating concurrent programs whose executions are simulated as test cases for the functional verification of the coherent shared-memory subsystem of a multicore chip. It emphasizes the principles defined by an MCM, with particular focus on coherence and consistency rules in Section 2.1. Additionally, Section 2.2 introduces the principles of a ML technique driven by interactions with its environment.

2.1 CACHE COHERENCE

In multicore chips, all cores share a common main memory for data exchange. However, accessing main memory incurs long latencies (often hundreds of cycles). To mitigate this cost, cache memories are introduced. These are smaller, faster storage units that may be private to a core or shared among several cores. While caches substantially reduce access latency, they also add complexity, as data becomes replicated: main memory preserves the global state, whereas each cache maintains a local state.

This complexity can be illustrated with a simple example. Table 1 shows two cores (C_1 and C_2) executing memory operations on the same location (X). For simplicity, this example assumes a *write-through cache*, where each store updates both the cache and the main memory. Assume that at time 0 the memory location X contains the value 0, and both private caches are empty. At times 1 and 2, C_1 and C_2 load X , respectively. Both read operations return the same value from main memory, which is also stored in their private caches for faster future access. At time 3, C_2 executes a store, updating its cache and the main memory (write-through caching policy). However, C_1 's cache is neither updated nor invalidated. As a result, the memory subsystem becomes incoherent: C_1 still holds the outdated value of X , while C_2 observes the updated value.

Table 1 – Illustrative example of an incoherent multicore architecture

Time	Event	Core C_1 cache content	Core C_2 cache content	Main Memory
0	<i>idle</i>	-	-	0
1	C_1 reads X	0	-	0
2	C_2 reads X	0	0	0
3	C_2 writes 1 into X	0	1	1

Source: Adapted from Hennessy & Patterson (2017)

A memory system is considered coherent if it satisfies two invariants (Nagarajan et al., 2020):

- **Single-Writer, Multiple Reader (SWMR)**: at any given memory location, at any specific moment, a memory location can either be accessed for writing and reading by a single core or be available for reading by multiple cores;
- **Data Value (DV)**: the value in a memory block at the start of a read-write epoch must be the same as the value at the end of the last read-write epoch of that block.

The example in Table 1 could illustrate a direct violation of these invariants if no action is taken to enforce coherence. When C_2 writes the new value of X at time 3, the SWMR invariant would be violated because C_2 assumes exclusive write access while C_1 still holds a stale copy, allowing inconsistent views of the same location. Similarly, the DV invariant would be broken, as C_1 would continue to observe the old value of X rather than the value established at the end of the previous epoch. To prevent such inconsistencies and ensure both invariants are upheld, multicore systems employ cache coherence protocols.

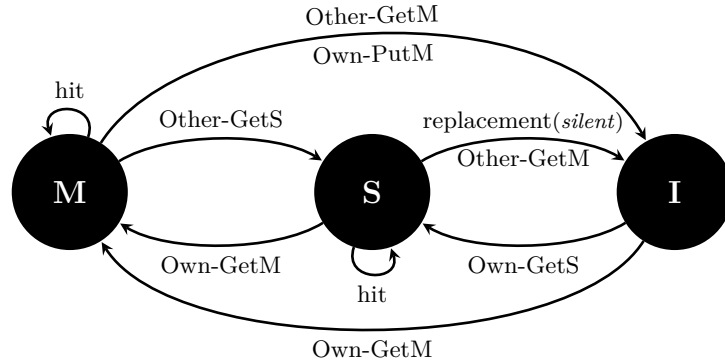
2.1.1 Coherence Protocols

Coherence protocols are implemented in hardware to enforce the invariants of cache coherence and, as a result, provide the abstraction of a coherent memory system. Under such protocols, all memory requests in a multicore system are handled by cache controllers. These controllers rely on an FSM to manage the states of cache blocks and arbitrate communication between cores. Each memory block is associated with a coherence state, recorded through status bits (in addition to other metadata such as validity or dirtiness). Two main classes of coherence protocols exist (Hennessy; Patterson, 2017): *snooping* protocols, in which caches monitor a broadcast medium to track block sharing, and *directory-based* protocols, in which block states are maintained in either a centralized or distributed directory.

One of the simplest coherence protocols is the Modified Shared Invalid (MSI) protocol, which defines three *stable* states. A stable state specifies the constraints under which a memory block resides. Transitions between stable states are not always direct; they may involve intermediate steps. Thus, a stable state can be understood as the condition of a block either before a transaction begins or after it completes. In contrast, a *transient state* represents an intermediate condition while a transaction is still pending between two stable states. For clarity, transient states are omitted from the following explanation. Figure 5 illustrates the MSI protocol and its three stable states:

- **Modified (M)**: read-write permissions (block is present only in a given cache);
- **Shared (S)**: read-only permission (block is present at least in one private cache);
- **Invalid (I)**: access revoked (block is not present in the cache or is obsolete).

Figure 5 – MSI protocol



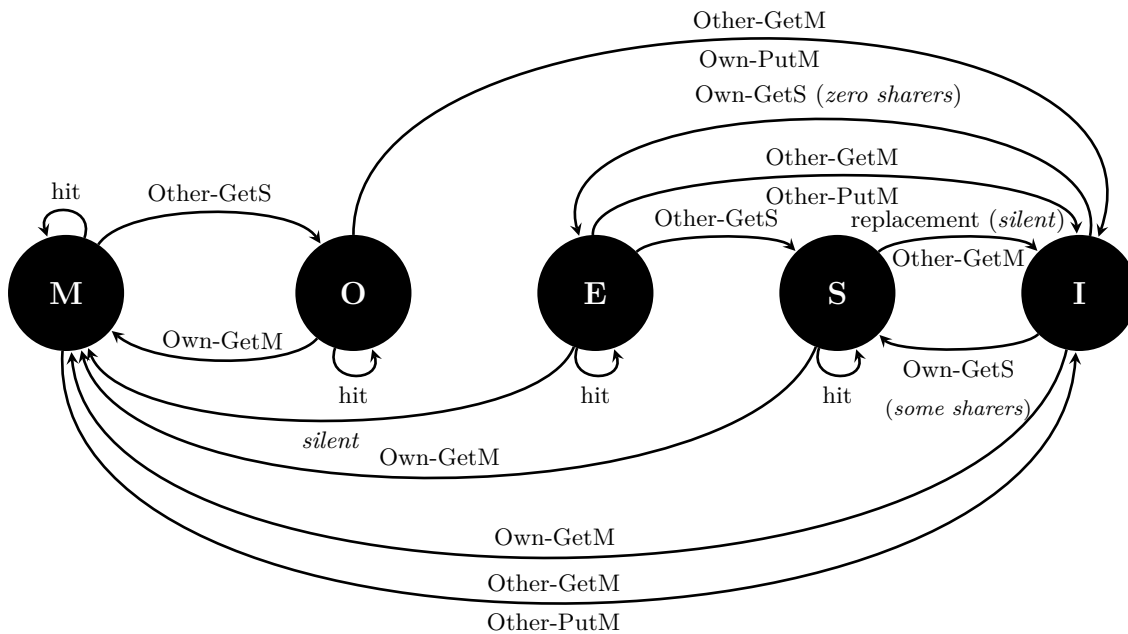
Source: Adapted from Nagarajan et al. (2020)

Transactions prefixed with *Other* are induced by remote accesses to the block, whereas those prefixed with *Own* originate from the local requester. A *hit* event denotes a successful read or write access to a cached block. The *GetM* transaction requests read-write permission to access a block, while the *GetS* transaction requests read-only permission. A *replacement* event occurs when an access targets a memory block that is not currently cached and maps to a fully occupied cache set. This transaction is silent, meaning it does not need to be reported to any cache controller.

The basic MSI protocol represents the minimal set of states required to enforce coherence, and serves as the foundation for several more advanced variants such as MESI, MOSI, and MOESI. These extensions introduce additional states and transitions to improve efficiency and reduce coherence traffic.¹ The MOESI protocol, illustrated in Figure 6, is a widely used extension of MSI that adds two states: *Exclusive* (E) and *Owned* (O). The *Exclusive* state (E) denotes that a block resides in exactly one cache and is clean. This property allows the cache controller to silently upgrade the block from state E to M when a store is issued, without the need for a *GetM* transaction. Therefore, it optimizes the common case of a read followed by a write to the same location, eliminating half of the coherence transactions. A block in this state transitions to *Shared* state (S) upon a remote read request, or to *Modified* state (M) upon a local write. The *Owned* state (O) state, in contrast, optimizes write operations when data is shared. The *Owned* state (O) denotes that a block has read-only permission, is dirty, and the local cache controller must respond to requests for that block. This property allows other caches to get read-only copies of the updated block without the need to write-back to main memory. Therefore, a single cache controller can hold the block in state O, while others can hold the block in state S. Since write-back can be deferred, this reduces unnecessary memory traffic.

¹ For instance, Intel’s i7 processors implement a variant of MESI called MESIF, which adds a *Forward* state (F) to designate the cache responsible for replying to a request when a block is shared among multiple cores (Hennessy; Patterson, 2017).

Figure 6 – MOESI protocol



Source: Adapted from Nagarajan et al. (2020)

2.1.2 Consistency Rules

Coherence rules enforce a consistent global view of memory accesses to the *same* shared location. In contrast, consistency rules define how accesses to *distinct* shared locations become visible to other threads and in what order. The Sequential Consistency (SC) model (Lamport, 1978) is the simplest MCM, requiring each core to preserve the program order of all memory accesses. This strict ordering, however, limits hardware optimizations. To improve performance, modern architectures often relax consistency rules, giving rise to models such as TSO, Partial Store Order (PSO), Weak Ordering (WO) and Relaxed Consistency (RC). While these relaxed models enable aggressive hardware optimizations, they also permit a much larger set of possible execution behaviors in parallel programs, which significantly increases the complexity of design validation.

Table 2 summarizes the four possible program-order relaxation scenarios involving reads (R) and writes (W) to distinct shared locations, together with examples of widely used architectures that implement them. The MCMs are organized from the most strict to the most relaxed, according to their consistency guarantees. To illustrate the key characteristics of those MCMs, consider the example in Figure 7. It shows a parallel program with two threads (*Thread 1* and *Thread 2*), each performing write operations. Capital letters denote shared memory locations, while lowercase letters represent register variables.

Table 2 – The relaxed order allowed for distinct memory models

MCM	Architecture	Relaxed Order			
		$R \rightarrow R$	$R \rightarrow W$	$W \rightarrow R$	$W \rightarrow W$
Sequential Consistency (SC)	several (optional)				
Total Store Ordering (TSO)	x86/SPARC			✓	
Partial Store Order (PSO)	SPARC			✓	✓
Weak Ordering (WO)	PowerPC	✓	✓	✓	✓
Relaxed Consistency (RC)	ARMv8/RISC-V	✓	✓	✓	✓

Source: Adapted from Hennessy & Patterson (2017)

Figure 7 – Illustration of valid outcomes from a parallel program

Thread 1	Thread 2
A = 0	B = 0
B = 1	A = 1
x = A	z = B

Source: Adapted from Hennessy & Patterson (2017).

Under SC, which strictly preserves program order, only the outcomes $(x = 0, z = 1)$, $(x = 1, z = 0)$, and $(x = 1, z = 1)$ are possible. The outcome $(x = 0, z = 0)$ would violate SC, as it contradicts the ordering constraints on memory operations. In contrast, under TSO, where the ordering of write-to-read ($W \rightarrow R$) is relaxed, the outcome $(x = 0, z = 0)$ becomes permissible. PSO allows the same outcome by introducing an even weaker memory ordering that relaxes write-to-write ($W \rightarrow W$) constraints. Although seemingly counterintuitive, this behavior is a direct consequence of the weaker guarantees provided by the chosen MCM. WO and RC differ primarily in how they handle synchronization: WO enforces global synchronization, while RC permits more flexibility by requiring synchronization only around specific data accesses.

2.1.3 Coverage Metrics

In simulation-based design verification, it is necessary to employ metrics capable of quantifying the extent to which a design's structure or behavior has been exercised during the verification process. This extent is referred to as *coverage*. In a cache-coherent multicore system, each level of the memory hierarchy employs an FSM responsible for implementing its portion of the coherence protocol. Accordingly, each cache instance, under the control of its cache controller, maintains an FSM specific to its hierarchy level. Since a coherence protocol is fundamentally defined by the interactions among the FSMs implemented by cache controllers at a given memory-hierarchy level, the implicitly specified system behavior can be *characterized* by the corresponding product FSM. This charac-

terization gives rise to the notions of *global* and *local* coverage.

Some DTGs exploit *global coverage* by reasoning over properties of the entire verification space induced by the product FSM (Wagner; Bertacco, 2008; Qin; Mishra, 2012; Lyu et al., 2019). However, capturing global coverage inherently limits their applicability across different coherence protocols, as such approaches depend on protocol-specific FSMs and their global interactions. In contrast, another class of DTGs focuses on *local coverage*, which improves reusability and requires a coverage space that grows linearly with the product of the number of processors and the number of private hierarchy levels (Elver; Nagarajan, 2016; Andrade et al., 2018; Andrade et al., 2020; Pfeifer et al., 2020). Nevertheless, local coverage has the inherent limitation of not directly capturing emergent behaviors arising from interactions among multiple processors and cache levels.

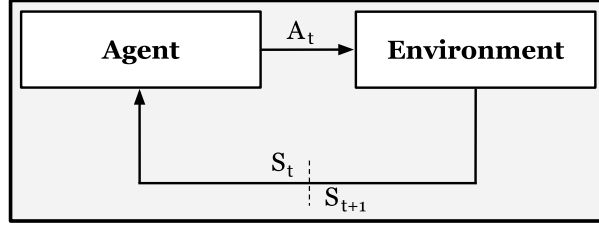
Within the scope of local coverage, *structural coverage* (Elver; Nagarajan, 2016) accounts for the execution of events in each FSM source code, independently of the number of cores in the system. Alternatively, *functional coverage* (Andrade et al., 2020) captures predefined interaction scenarios across hierarchical levels, such as requests propagated from a requester to the directory and subsequently to a sharer or owner. Because coverage goals distinguish identical transitions occurring in different core domains, the number of transitions to be covered increases proportionally with the number of cores at each private cache level.

2.2 MACHINE LEARNING (ML)

ML has been successfully applied across a wide range of domains, demonstrating its versatility and effectiveness in solving complex problems. Among the many applications of ML, the underlying techniques are often categorized according to how models learn from data. In *supervised learning*, models are trained on labeled datasets, where the correct output is provided for each input, enabling tasks such as classification and regression. In contrast, *unsupervised learning* operates on unlabeled data, seeking to uncover hidden structures or groupings, as in clustering or dimensionality reduction. A third paradigm, RL, is driven by interactions with an environment: an agent learns to make decisions by receiving feedback in the form of rewards or penalties.

As discussed in Chapter 1, DTG approaches employ an RTG engine orchestrated by a *Directing Engine* during the pre-silicon verification phase. Each generated test is executed on a simulated DUV, and the resulting memory events are collected for analysis by an MCM checker. Because of its effectiveness in addressing sequential decision-making problems, RL is particularly well-suited for guiding DTG. A brief overview of RL is provided in the next section.

Figure 8 – The agent–environment interaction in an MDP.



Source: Adapted from Sutton & Barto (2018).

2.2.1 Reinforcement Learning (RL)

The RL approach addresses the Markov Decision Process (MDP) illustrated in Figure 8, which consists of an *agent* interacting with an *environment*. The environment provides an abstracted model of real-world behavior, while the agent operates in discrete time steps $0, 1, 2, \dots, t$. The dotted line in Figure 8 represents a temporal boundary, marking the evolution between successive steps. At a given time step t , the agent observes the current state of the environment (S_t) and reacts by taking an action (A_t). At the following step $t + 1$, the environment transitions to a new state (S_{t+1}) and provides the agent with a reward signal (R_{t+1}). This interaction can be described as the following sequence (a.k.a *trajectory*): $S_0, A_0, R_1, S_1, A_1, R_2, S_2, A_2, R_3, \dots$

The primary goal of RL is to maximize the cumulative *reward* signal over time, as defined by the environment. A *policy* specifies the agent’s behavior by mapping states to actions. A *value function* estimates the expected long-term return of a given policy and is central to policy improvement in classic algorithms (Bellman, 1957; Watkins; Dayan, 1992; Rummery; Niranjan, 1994). Rewards and value functions are complementary: rewards quantify the immediate benefit of an action, while value functions capture its expected long-term benefit. An MDP is finite when the sets of states $\mathcal{S}$, actions $\mathcal{A}$, and rewards $\mathcal{R}$ are finite. The stochastic transition function (Equation 2.1) defines the probabilistic relationship between states, actions, and rewards (Sutton; Barto, 2018).

$$p(s', r | s, a) \doteq \Pr\{S_{t+1} = s', R_{t+1} = r | S_t = s, A_t = a\}, \quad \forall s', s \in \mathcal{S}, r \in \mathcal{R}, a \in \mathcal{A}(s) \quad (2.1)$$

Ideally, a state should encode all the information about the environment necessary to predict its future behavior. When this condition is met, the state is said to satisfy the *Markov property*. In practice, however, agents often have only partial observations of the environment. Consequently, techniques are required to approximate a Markovian state representation from these observations. Such techniques must also address the

well-known *exploration–exploitation dilemma* (Robbins, 1952). On one hand, *exploitation* leverages the agent’s current knowledge to maximize immediate rewards; on the other hand, *exploration* deliberately seeks uncertain or less-known actions to improve the policy and increase long-term returns.

2.2.2 Deep Q-Learning

Q-learning (Watkins, 1989; Watkins; Dayan, 1992) is an off-policy temporal-difference control algorithm that estimates the action-value function $Q(s, a)$ and converges, under suitable conditions, to the optimal action-value function Q^* . Its update rule (Equation 2.2) adjusts estimates based on observed transitions, independently of the behavior policy used to generate experience. The step-size parameter, or learning rate, $\alpha \in (0, 1]$, and the discount factor $\gamma \in (0, 1]$, are hyperparameters that regulate, respectively, how much new information overrides past estimates and how future rewards are weighted relative to immediate ones.

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha \left[R_{t+1} + \gamma \max_a Q(S_{t+1}, a) - Q(S_t, A_t) \right] \quad (2.2)$$

In Q-learning, the behavior policy determines which state–action pairs are explored and updated. Convergence to the optimal action-value function Q^* is guaranteed provided that all state–action pairs are visited infinitely often and the learning rate is appropriately decayed over time (Sutton; Barto, 2018). While Q-learning is theoretically applicable to a broad range of problems, real-world domains frequently involve high-dimensional state spaces, which significantly slows convergence. A common strategy to mitigate this issue is to approximate the Q-function with non-linear function approximators. Neural networks, in particular, are well-suited for this purpose, as they can efficiently process high-dimensional sensory inputs from diverse domains.

The DQN algorithm proposed by Google DeepMind (Mnih et al., 2013) extends Q-learning by using a Deep Neural Network (DNN) to approximate the action-value function. To improve sample efficiency and stabilize training, it employs *experience replay* (Lin, 1992), which stores past transitions in a *replay buffer* and samples them randomly during updates, thereby reducing correlations between consecutive observations. Remarkably, DQN achieved human-level performance on a variety of Atari games directly from raw pixel inputs, without requiring handcrafted features. To address the exploration–exploitation dilemma, DQN typically adopts an ϵ -greedy strategy: with probability $1 - \epsilon$ the agent selects the action with the highest estimated value (*exploitation*), and with probability ϵ it selects a random action (*exploration*). The exploration rate ϵ may be fixed or annealed over time.

Subsequent research introduced several extensions to DQN that significantly enhanced its stability and performance. A notable example is *Rainbow* (Hessel et al., 2018), which unified six key improvements, including Double Q-learning (Hasselt; Guez; Silver, 2016), Dueling Networks (Wang et al., 2016), Prioritized Experience Replay (Schaul et al., 2016), and additional advances such as Distributional Reinforcement Learning (Belle-mare; Dabney; Munos, 2017), Multi-step Returns (Mnih et al., 2016), and Noisy Networks for exploration (Fortunato et al., 2017). While these optimizations represent important milestones, they are not further detailed here, as they lie beyond the scope of this thesis. It is worth noting, however, that the state-of-the-art RLG (Pfeifer et al., 2020) and the prototype LBG (proposed in this thesis) incorporate all of these optimizations, whereas DQG relies solely on the original DQN formulation.

3 RELATED WORK

When operations from distinct threads collide at a same shared location and are not synchronized, the resulting non-determinism often induces more than a legal value for a given shared location (Saha et al., 1995; Adir; Shurek, 2002). Therefore, it is impossible to build a golden model to predict the expected results for an arbitrary concurrent program. A pragmatic solution is to generate each test carrying its own expected results, i.e. a *self-checking test*.

However, if a generator selects instructions arbitrarily for each thread, the resulting concurrent program produces race conditions, making the results unpredictable and self-checking impossible. That is why early approaches prevented collisions to avoid unpredictable outcomes (Sullivan et al., 1995). Since collisions are key to exposing bugs, late approaches relied on value prediction to enable racy self-checking tests (Adir; Shurek, 2002).

The limitations of race-free self-checking tests and the complexity of predicting values for racy self-checking tests can both be avoided when verification relies on an external MCM checker (Saha et al., 1995; Hangal et al., 2004; Manovit; Hangal, 2006; Roy et al., 2006; Shacham et al., 2008; Hu et al., 2012; Graf; Andrade; Santos, 2023), because generators can synthesize *tests without results*. This largely simplifies test generation, and yet leads to effective tests with intensive collisions.

This chapter reviews related work on test generation for functional verification of shared memory behavior. It first considers constrained RTG techniques, illustrating how biasing constraints can significantly enhance verification quality (Section 3.1). It then analyzes two principal classes of DTG techniques: model-based and data-driven generators (Section 3.2). Finally, Section 3.3 discusses the limitations of prior methods and outlines how these motivate the contributions of this thesis.

3.1 RANDOM TEST GENERATION (RTG)

RTG is the usual approach to test generation for post-silicon validation. In that context, it is sufficient to constrain program properties, such as the number of memory operations, the number of shared locations, and the number of threads, because post-silicon validation can afford running long tests with millions of memory operations to reach proper coverage.

A few early works on functional verification have either reused legacy RTG techniques directly (Shacham et al., 2008) or proposed slight variations of it (Rambo; Henschel; Santos, 2011), but they were limited to *conventional constraints* capturing program properties. Since pre-silicon tests are limited to tens of thousands of operations, sophisticated RTG techniques proposed the use of *biasing constraints* to exploit shared memory properties that increase coverage while keeping tests short enough.

Table 3 – Constraint exploitation for distinct RTG approaches.

Work	Constraint Exploitation		
	Conventional	Collision	Eviction
(Shacham et al., 2008; Rambo; Henschel; Santos, 2011)	✓		
(Adir; Shurek, 2002; Adir et al., 2004)	✓	✓	✓
(Andrade; Graf; Santos, 2016)	✓	✓	
(Andrade; Graf; dos Santos, 2019)	✓	✓	✓

The exploitation of biasing constraints was first proposed by IBM in the scope of the Genesys-Pro verification framework (Adir et al., 2004). In that approach, test generation is cast as an Constraint Satisfaction Problem (CSP). Although early versions of that framework did not target non-deterministic behaviors, it was later extended to enable the generation of collision¹ scenarios (Adir; Shurek, 2002), because programs with intensive data races tend to expose design errors faster (Hangal et al., 2004; Shacham et al., 2008).

An early work (Andrade; Graf; Santos, 2016) exploited non-conventional constraints on RTG so as to induce canonical multiprocessor dependence chains across threads (Gharachorloo, 1995). Indeed, the structure of those chains serves as biasing constraints that foster collision-induced state transitions. A later work generalizes that early chaining technique and proposes another type of biasing constraint to foster eviction-induced state transitions (Andrade; Graf; dos Santos, 2019). Experimental results show significant coverage improvement when both types of biasing constraints are imposed simultaneously. Table 3 summarizes how distinct approaches exploit constraints on RTG.

3.2 DIRECTED TEST GENERATION (DTG)

There are two basic approaches to DTG (Fine et al., 2005). *Model-based generators* rely on some representation of coverage goals (such as a state machine, a graph, or an equation), which either explicitly encodes coverage requirements (such as the states and transitions of a coherence protocol) or implicitly captures how to improve coverage (such as the expected variation in coverage as a function of the number of operations and the amount of shared locations). *Data-driven generators* rely on a feedback loop to dynamically modify test generation parameters by means of some meta-heuristic or ML technique that monitors the environment and responds to the observed coverage evolution. For convenience, Table 4 labels the main works on DTG that will be analyzed in the following sections.

Table 5 classifies each DTG technique shown in Table 4 as model-based or data-

¹ The usual terminology says that two memory operations *collide* when they reference the same shared location. If two memory operations collide, and at least one of them is a store, then the usual terminology says that they *conflict*.

Table 4 – Adopted acronyms for existing DTG techniques.

Acronym	Name	Reference
BNG	Bayesian Network Generator	(Fine; Ziv, 2003; Fine et al., 2005)
Mcjammer	Mcjammer Test Generator	(Wagner; Bertacco, 2008)
MTG	McVerSi Test Generator	(Elver; Nagarajan, 2016)
ETG	Euler Tour Generator	(Qin; Mishra, 2012)
SCG	Steep Coverage Generator	(Andrade et al., 2018)
QSG	Quotient Space Generator	(Lyu et al., 2019)
RLG	Reinforcement Learning Generator	(Pfeifer et al., 2020)
HTG	Hybrid Test Generator	(Andrade et al., 2020)

driven. Notice that one technique combines model-based and data-driven approaches. Table 5 also indicates the key mechanism of each technique and whether or not it relies on an RTG engine. Note that some techniques do *not* require an RTG engine to synthesize tests.

3.2.1 Model-based Generators

The Euler Tour Generator (ETG) proposed by Qin & Mishra (2012) works upon a graph representation of the product FSM, which explicitly enumerates the entire protocol space. It exploits topological properties of the graph to synthesize a test suite that induces an efficient Euler tour of its edges. Since no transition is ever visited more than once, the test suite covers all state transitions. Experimental results indicate that the explicit enumeration of the full state space remains feasible for single-level protocols involving up to 16 cores. However, the approach is inherently limited by the exponential growth in transition count with the number of cores, thus deserving elaboration. The Quotient Space Generator (QSG) proposed by Lyu et al. (2019) relies on a symmetry reduction technique that decomposes the state space into equivalence classes and narrows down the explored state space to those classes. This allows a trade-off between coverage and verification time. However, the technique is unsuitable for verifying memory consistency,

Table 5 – Model-based, data-driven, and hybrid approaches to DTG.

Approach	Class		Mechanism	
	Model-based	Data-Driven	Key	RTG Engine
BNG		✓	Bayesian Network	✓
Mcjammer		✓	Multiple intelligent Agents	
MTG		✓	Genetic Programming	
ETG	✓		Euler Tour	
SCG	✓		Steep Ascent	✓
QSG	✓		Quotient Space	
RLG		✓	Reinforcement Learning	✓
HTG	✓	✓	Steep Ascent + Neighborhood Search	✓

because it requires artificial order constraints that inhibit data races. The approach limits error discovery because it leads to abstract transitions that may aggregate multiple paths over transient states, but only one of them is covered in the Euler tour.

The Steep Coverage Generator (SCG) proposed by Andrade et al. (2018) employs a directing engine that modifies the parameters of an RTG engine. The directing engine implicitly relies on a coverage model that induces a sequence of settings for a triple of parameters (n, s, k) leading to steep coverage ascent. That sequence is induced by alternately fostering collision and eviction-induced transitions. The directing engine can command such alternation because the RTG engine supports (collision and eviction) biasing constraints.

3.2.2 Data-driven Generators

The Bayesian Network Generator (BNG) proposed by Fine et al. (2005) employs statistical inference to dynamically redefine directives for an RTG engine. The approach is based on modeling the relationship between coverage information and test directives as Bayesian networks. As a result, coverage directed test generation is cast as a process which is performed in two steps. In the first step, a training set is used to learn the parameters of a Bayesian network that models the relationship between coverage and directives. In the second step, the Bayesian network is used to obtain the most probable directives that would lead to a given coverage value. This approach admittedly requires understanding of the type of knowledge that should be encoded in the model.

The Mcjammer generator proposed by Wagner & Bertacco (2008) is a scalable approach that avoids the enumeration of the full protocol space. Instead of a centralized directing engine, it relies on distributed intelligent agents. Each agent has a simplified view of the design under verification, which is called Dichotomic Finite State Machine (DFSM), and it captures the protocol behavior from the perspective of a given core domain. A state in the DFSM captures the state of a block in a local cache and an aggregation of the state of that block in the caches from other domains. A DFSM only allows the agent to distinguish between its own actions and those of the environment (which includes the other agents) and has its own internal representation of coverage. Agents keep track of DFSM transitions visited in the past and use that information to direct testing toward the unexplored ones for higher transition coverage. The generator is reusable only for derivative designs that comply with the same protocol, because the DFSMs must be modified for porting the generator to a protocol variant.

The MTG proposed by Elver & Nagarajan (2016) relies on Genetic Programming (GP) to synthesize a suite of tests. Each test is represented by a chromosome, where each gene corresponds to a memory operation. It relies on a largely unconstrained RTG engine only for the creation of an initial population of tests. A fitness function ranks the tests of a population according to some coverage metric defined by the verification environment.

From the fittest tests of a previous population, the directing engine performs crossover to obtain the off-springs that will form a new generation. MTG employs a selective crossover that favors the choice of genes (memory operations) that highly contribute towards non-determinism. All the tests of a suite are limited to have a constant number of operations, although the number of operations per thread may vary. The number of shared locations may vary from one test to another, but the number of locations mapping to the same cache line is fixed for all tests.

The RLG proposed by Pfeifer et al. (2020) uses an agent that directs the engine, which interacts with the generation environment. It employs actions that modify the parameters of an RTG engine. Originally, the actions modified three parameters: the number of operations (n), the number of shared locations (s), and the number of locations mapping to the same cache line (k). In a later version (Pfeifer, 2021), the actions were generalized to control a parameter that defines whether false sharing is allowed. The generated test suite consists of tests with possibly different numbers of operations, distinct numbers of locations, and different numbers of locations mapped to the same cache line.

3.2.3 Hybrid Generation

The HTG proposed by Andrade et al. (2020) is a generalization of the already described model-based SCG generator that includes a data-driven engine. The new engine explores neighborhoods of the RTG space. Instead of a plain composition of well-known techniques, the approach was designed for proper decoupling of abilities: data-driven *exploration* of the generation space and model-based *exploitation* of biasing constraints. The generated test suite consists of tests with possibly different numbers of operations, distinct numbers of locations, and different numbers of locations mapping to the same cache line.

3.3 DISCUSSION

Since multicore chips are pervasive across computing classes with distinct requirements, designs have to implement different MCMs and distinct protocols with dissimilar microarchitectures. Therefore, a new DTG technique should be largely independent of the DUV.

Let us consider a hypothetical example to illustrate the importance of DUV independence. Assume that an open architecture such as RISC-V would be used for two different multicore designs: one targeting a high-end embedded application and another targeting a domain-specific architecture. Their coherence protocols might be quite different: for instance, a 1-level MESI protocol in one case and a 2-level MOESI protocol in another. Since both designs rely on the same ISA, the MCM is the same. However, if one of such designs happened to rely on ARMv8, their MCMs would be different.

The hypothesis driving the research of this thesis is that “*it is possible to build a directed test generator which is largely independent of the DUV and which exhibits faster coverage evolution as compared to state-of-the-art generators, as a result of an improved DUV observability and a proper balance between exploration of the generation space and exploitation of biasing constraints*”. The rationale underlying such a hypothesis is that proper exploration should avoid both limiting coverage and generating tests that hardly contribute to coverage improvement. That is why this section analyses the limitations of related works with respect to their capabilities of providing DUV independence, observability, proper space exploration, and adequate constraint exploitation.

3.3.1 Limited DUV Independence

MCM independence. Since MTG, SCG, HTG, and RLG are all techniques relying on an external MCM checker, they are obviously independent of the target MCM. Mcjammer and the approaches by Qin & Mishra (2012) and by Lyu et al. (2019) aim at protocol verification. Therefore, they are unable to check shared memory behaviors that are not captured by the protocol FSM.

Protocol independence. MTG, SCG, HTG, and RLG are all independent of the target coherence protocol, as opposed to Mcjammer and the approaches by Qin & Mishra (2012) and by Lyu et al. (2019), where tests are tailored to cover the state transitions of the target protocol.

Thus, for large independence of the DUV, a novel approach to DTG should rely on an external MCM checker.

3.3.2 Limited Exploration of the Test Generation Space

The main limitation of some DTG techniques is that they rely on RTG for random operation selection and random address assignment (Andrade et al., 2018; Andrade et al., 2020; Pfeifer et al., 2020). Such RTG-based DTG techniques rely on generation parameters to constrain, for instance, the number of operations (n) and the number of shared locations (s). For the same assignment of values to the parameters (n, s), distinct random tests can be generated depending on the selected random seed, each leading to distinct coverage. Therefore, no coverage control exists on distinct tests generated with the same setting (n, s). To try to compensate for the lack of coverage control, multiple seeds are used to induce as many random tests, and we can only hope that their cumulative coverage is adequate.

Although the parameter space is a compact representation of the generation space, it does not represent a proper test representation, because the same assignment of

values to the generation parameters may correspond to tests of quite different coverage values, as shown by Andrade, Graf & dos Santos (2019).

There are, however, two notable exceptions. First, the approach proposed by Qin & Mishra (2012) does not rely on RTG at all, but reaching full coverage requires explicit enumeration of state transitions, which is not scalable with growing core counts. That is why the approach deserved a later improvement by Lyu et al. (2019).

Second, the MTG approach by Elver & Nagarajan (2016) relies on random tests only for building the initial population. Since it relies on GP, MTG requires some encoding for performing genetic operators such as mutation and crossover. MTG's representation requires a constant (overall) number of operations, although the number of operations per thread is not necessarily constant. As a result, the entire suite generated with GP consists of tests of exactly the same size. As shown by Andrade et al. (2020), short tests should be used as far as cumulative coverage grows (to save runtime) for proper coverage evolution. Otherwise, progressively larger tests should be run (to obtain extra coverage at the expense of runtime). Therefore, an encoding leading to a suite of tests with fixed test sizes is likely to limit coverage evolution.

3.3.3 Limited Exploitation of Constraints

The test representation adopted by MTG captures operation type (load or store) and shared location, but it does not capture the mapping of distinct locations to the same cache line. This makes the encoding unable to control cache eviction. That is why, before launching test generation, an ad-hoc biasing constraint is applied to stimulate eviction. However, this is a rather limited mechanism, because all tests of the entire suite will have similar eviction patterns, which limits their individual contribution to the cumulative coverage.

In contrast, SCG, HTG, and RLG all apply distinct biasing constraints to each test of a suite, leading to superior coverage evolution. By setting distinct values to a parameter (k), those DTGs command an underlying RTG engine to impose different biasing constraints to each test. Although they have much better control on eviction than MTG, none rely on a test encoding but on generation parameters.

3.3.4 The Lack of a Proper Test Representation

Quite distinct test representations have been used by DTG approaches reported in the literature. A general representation based on disjoint directed acyclic graphs (Corno; Cumani; Squillero, 2003) was used to build the McVerSi Test Generator (Elver; Nagarajan, 2016), which relies on GP. A representation based on a constraint network (Adir et al., 2004) was used for building a generator based on Bayesian Networks (Fine; Ziv, 2003).

Being too general, such representations cannot directly encode shared-memory behaviors, as desirable for defining effective actions for an agent.

Specific test representations targeting shared-memory behavior have been reported. Generators based on an Euler tour mechanism (Qin; Mishra, 2012; Lyu et al., 2019) cast the entire state space of a coherence protocol into a state machine graph. Their underlying test representation is a path in that graph. Albeit adequate for inducing coherence behavior for a given shared location, such path-based representation is not suitable for inducing behaviors involving distinct shared locations (as required by MCMs). On the other hand, test representations adopted for RTG (Hangal et al., 2004; Manovit; Hangal, 2006) and for DTG (Andrade; Graf; Santos, 2016; Andrade et al., 2020; Pfeifer et al., 2020) often consist of tuples of parameters, used as directives for commanding an RTG engine. However, representations based on RTG directives have an inherent drawback: the same directive may randomly induce different tests leading to quite distinct coverage (Fine; Ziv, 2003).

To avoid that randomness might impair learning (thereby limiting coverage), the conversion of a test encoding into an actual test program should be *deterministic*, and each encoding should produce a *unique* test. From the analysis of the literature, it can be concluded that there is a lack of proper test representation to drive DTG. Ideally, such a test should:

1. Not be tied to the target protocol (as in Mcjammer and in the approaches by Qin & Mishra (2012) and by Lyu et al. (2019));
2. Capture not only operation type and shared location (as MTG does), but also the mapping of distinct locations to the same cache line (unlike MTG), instead of relying on an RTG parameter for cache eviction control (as SCG, HTG, and RLG do with the parameter k);
3. Capture variable test sizes (unlike MTG), instead of relying on an RTG parameter for size control (as SCG, HTG, and RLG do with the parameter n); and
4. Not be tied to the technique adopted to direct test generation (unlike MTG, whose encoding is tied to the needs of genetic operators).

The proposed test representation, which is the first contribution of the thesis, is discussed in the Chapter 4. It provides provides fine-grain control on relevant memory events and is independent of the mechanism used in the Directing Engine (GP, RL, or any other mechanism).

3.3.5 The Lack of a Proper DUV Observation

Although existing data-driven generators rely on different engines to guide test generation, they are primarily directed by *coverage*. While coverage is useful for tracking the progress of the verification process, it does not constitute a direct observation of

the environment. In particular, it is not an injection: the same coverage fraction may correspond to different subsets of goals with identical cardinality but distinct verification significance. To achieve more effective guidance, DTG requires additional observations from the DUV, which are typically limited to external checkers. By exploiting such domain-specific information, a generator can accelerate coverage evolution and improve overall verification efficiency.

The analysis of the literature reveals a lack of suitable observation mechanisms for recovering state information across different DUVs. Ideally, such an observation should:

1. Not depend on a particular MCM, protocol, or microarchitecture, thereby avoiding coupling to a specific DUV;
2. Not rely on handcrafted features, but instead exploit raw memory data;
3. Capture fundamental shared-memory behavior rather than relying on partial coverage metrics; and
4. Leverage generation constraints to ensure a *unique representation* for every observable execution induced by a test suite.

The second contribution of this thesis, presented in Chapter 5, is a domain-specific multicore state representation. This representation enables the recovery of state information directly from raw environmental observations, allowing agents to learn through interaction how to improve coverage for any verification task in the target domain without relying on task-specific handcrafted features. Furthermore, it encodes the complete interaction history that may be relevant to the agent’s decision-making process.

4 A CANONICAL TEST REPRESENTATION FOR VERIFICATION OF SHARED-MEMORY BEHAVIOR IN MULTIPROCESSOR SYSTEMS

This chapter reproduces the article: *Bruno D. Miranda, Márcio Castro, and Luiz C. V. dos Santos. A Canonical Test Representation for Verification of Shared-Memory Behavior in Multiprocessor Systems. ACM Transactions on Design Automation of Electronic Systems (TODAES), ACM, 2025. DOI: 10.1145/3762184.* It introduces a novel canonical representation for test programs targeting the verification of shared-memory behavior in multiprocessor systems. The proposed representation ensures both uniqueness and completeness of test encodings, preventing redundant test generation and avoiding limitations in test suites. It was experimentally validated with a Reinforcement Learning-based generator, which achieved higher coverage and faster error diagnosis than state-of-the-art alternatives in most verification scenarios.

4.1 INTRODUCTION

In a multiprocessor system, processors interact through shared memory. Each interaction must obey the shared memory behavior specified by a MCM (Adve; Gharchorloo, 1996; Nagarajan et al., 2020), which constrains the ordering of load and store operations. Such ordering may involve accesses to the same or to distinct shared locations. Accesses to the same location must obey coherence and atomicity rules. Accesses to distinct locations must obey rules for relaxation of program order. The state space of shared-memory behavior is huge. It increases with core count and with relaxation of program order and atomicity. This leads to a challenging problem for which many approaches exist, no single approach seems good enough to find all bugs, and none seems suitable to all design phases. The scope of this chapter is the design verification of a multicore chip or multichip multiprocessor by running concurrent test programs until predefined goals are reached.

4.1.1 Verification requirements and goals

In general, simulation-based design verification requires some metric to evaluate the fraction of the design structure or behavior that has been validated during the verification process. This fraction is called *coverage*. Therefore, one major verification goal is how to generate stimuli leading to high coverage. Besides, design verification requires some technique to check whether or not the observed outcome meets the expected result according to some golden model. Therefore, one major verification requirement is accurate *error diagnosis*. The ultimate goal of simulation-based verification is to attest that no design error was diagnosed until full coverage is reached or until some time limit is attained.

Beyond such general requirements and goals, the verification of the shared-memory subsystem has some specific issues. Instead of directly synthesizing input stimuli to check a multicore design, test generation usually synthesizes test *programs* to indirectly induce proper input stimuli when executed. Test programs for multicore designs are concurrent programs, and therefore the MCM allows multiple behaviors for the same program, leading to multiple possible outcomes for the same input stimuli. Thus, there is no one-to-one golden model for the expected behavior, as it is usual in other verification domains. At first glance, this issued could be handled by restricting test generation to produce synchronized programs, but this would actually limit verification. Since synchronization primitives are intended to avoid data races, they end up preventing collisions of memory operations from distinct threads, and collisions are crucial to foster coverage and error diagnosis (Adir; Shurek, 2002; Elver; Nagarajan, 2016; Shacham et al., 2008).

There are two general methods to overcome the lack of a one-to-one golden model.

- The generator produces *self-checking tests* (Adir et al., 2004; Adir; Shurek, 2002), and the expected results for each test program have to be predicted by the test generator with basis on the MCM rules, but value prediction of non-unique results largely adds to the generation complexity.
- The generator produces *tests without results* (Saha et al., 1995; Hangal et al., 2004; Manovit; Hangal, 2006; Roy et al., 2006; Shacham et al., 2008; Hu et al., 2012; Graf; Andrade; Santos, 2023), and its main role is to improve coverage, while the role of error diagnosis is performed by an *external MCM checker*.

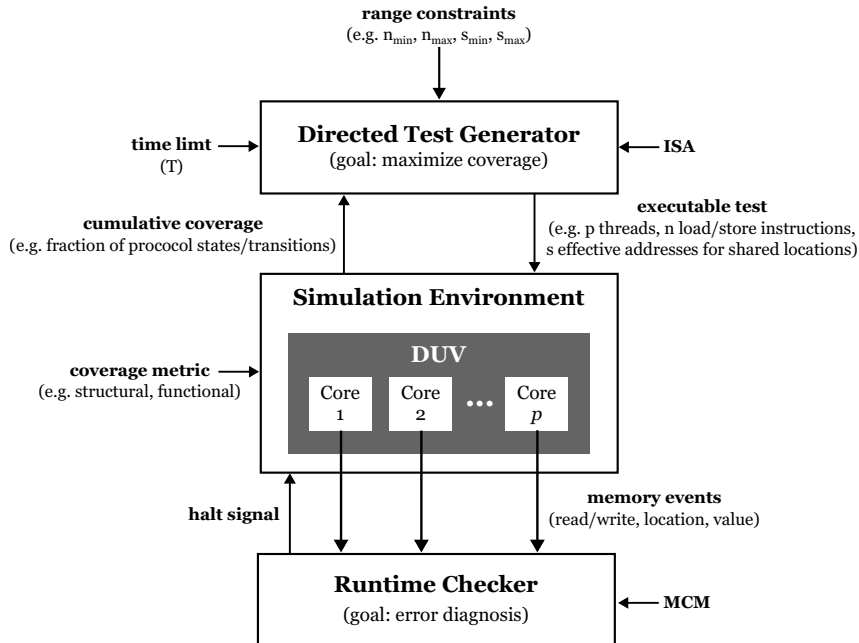
The approach proposed in this chapter follows the second method. The goal of the test generator is to improve coverage, and it assumes an independent runtime checker that verifies whether or not the shared-memory behavior induced by a generated test program meets the rules of a given MCM. If an error is detected on-the-fly, the checker stops simulation, and reports it.

4.1.2 Main approaches to test generation

Constrained Random Test Generation (RTG) has long been the conventional approach to avoid the enumeration of the interactions between processors through shared memory. When each thread is independently generated at random (under range constraints), the probability of covering all interactions increases with thread size (Wood; Gibson; Katz, 1990). That is why RTG is very efficient for multicore chip testing, because large test programs can directly execute in the hardware prototype.

For design verification, however, since test programs execute much slowly on a simulation of the design representation, they must be shorter. To reach proper coverage with shorter programs, Directed Test Generation (DTG) is preferred to RTG. Learn-

Figure 9 – A typical verification framework when using a test generator and an external checker



ing agents have been envisaged for DTG (Ioannides; Eder, 2012), and their formulation requires a proper representation for a test program.

4.1.3 Methodology and high-level ideas

Fig. 9 illustrates the generic components of the verification environment used in most approaches adopting directed test generation. A generator produces an executable test program for a given *instruction-set architecture* (ISA). That program is executed within a simulation environment containing a design representation of the multicore *device under verification* (DUV). Such execution induces stimuli at the interfaces between each core and the shared-memory components (e.g. private cache controllers and directory controller). A runtime checker, which is fed with the rules of an MCM, relies on monitors placed at the shared-memory interface for sampling memory events to be analyzed on-the-fly. If the memory events do not comply with the order specified by the MCM or if there is some inconsistency between written and read values, the checker issues a halt signal to stop simulation, and it reports an error.

If the executed test did not expose any error, the simulation environment evaluates the contribution of that test to coverage according to a given *metric*, and adds it to the cumulative coverage. Given a coverage value, the generator synthesizes another test, and a new cycle of verification is launched until some time limit is attained, or full coverage is reached, or an error is found.

In such a verification environment, the goal of the directed test generator is to produce a sequence of tests, i.e. a *test suite*, such that the cumulative coverage is maximal within the available time limit.

The partitioning of the main verification tasks into distinct engines has pragmatic consequences for directed test generation. For a given MCM, the runtime checker is independent of coherence protocol (either snooping or directory-based variants), and it is designed to be largely independent of micro-architecture optimizations (e.g. write buffers, read bypassing, lockup-free caches, etc). Since the runtime checker is the engine in charge of the analysis of memory events under the adopted methodology, directed test generation is *independent* of which MCM, coherence protocol, and micro-architecture optimizations are chosen for the target DUV.

Usually, the techniques used to build directed test generators and MCM runtime checkers are designed to scale gracefully with increasing core counts. Besides, the time to generate a test and the time to check if the behavior it induces satisfies the MCM are both *negligible* as compared to the simulation time required to execute that test. Indeed, it is exactly such dominance of *simulation time* that motivates the investigation of novel techniques to improve the *quality of the generated tests* so as to reduce the number of tests required to reach high coverage and proper error diagnosis, thereby reducing the overall simulation time for the verification process. Different mechanisms can be used by the directed test generator to improve the quality of the tests, such as Genetic Programming (Elver; Nagarajan, 2016) and Bayesian Networks (Fine; Ziv, 2003).

This chapter casts the target verification problem into a Reinforcement Learning framework. An *agent* inside the directed test generator takes successive *actions* that synthesize the sequence of programs of a test suite. As a result, the contribution of each test program to coverage can be seen a *reward signal*, and the agent can learn from interaction with the simulation *environment*. The goal of the agent is to learn how to produce a test suite that maximizes the expected return in the long run, i.e. the cumulative coverage. The main novelty of this chapter is a way to represent test programs that allows the design of efficient actions for the target verification domain, as follows.

4.1.4 Contribution

This chapter proposes a novel test-program representation for a specific application domain: the verification of multiprocessor shared-memory behavior when test generation is combined with MCM checkers (Saha et al., 1995; Hangal et al., 2004; Manovit; Hangal, 2006; Roy et al., 2006; Shacham et al., 2008; Hu et al., 2012; Graf; Andrade; Santos, 2023). It is intended as one of the main keys to the design of new test generators, especially when directed by learning agents. It encodes necessary conditions for primal behaviors to be induced at runtime.

For encoding a test, we rely on different collections of memory operations, each induced by a distinct equivalence relation. We prove the uniqueness of each encoding and the completeness of the representation.

To provide experimental evidence of the effectiveness of our representation, we

have built a generator directed by a Reinforcement Learning agent, and we have designed its actions with basis on our encoding. To properly evaluate the power of the proposed representation, we lowered the power of the agent. We deliberately tailored the agent to be minimalist: only three categories of actions were defined. Even with such a reduced set of actions, the generator based on our representation was able to reach the largest coverage and the fastest error diagnosis in 3/4 of the scenarios, as compared to sophisticated directed test generators (Elver; Nagarajan, 2016; Pfeifer et al., 2020; Andrade et al., 2020).

The remainder of this chapter is organized as follows. Section 4.2 reviews the use of test representations for DTG. Section 4.3 describes our representation, shows an illustrative example, and gives proven guarantees. Section 4.4 casts DTG as an optimization problem. Section 4.5 applies our representation to the building of a learning-based test generator for solving that problem. Section 4.6 compares it with state-of-the-art generators. Section 4.7 draws the main conclusions and opens perspectives for the design of future generators.

4.2 RELATED WORK

This section first reviews features of shared-memory tests, and then discusses test-program representations.

When operations from distinct threads collide at a same shared location and are not synchronized, the resulting non-determinism often induces more than one legal value for a given shared location (Saha et al., 1995; Adir; Shurek, 2002). Therefore, it is impossible to build a golden model to predict the expected results for an arbitrary concurrent program. A pragmatic solution is to generate each test carrying its own expected results, i.e. a *self-checking test*.

However, if a generator selects instructions arbitrarily for each thread, the resulting concurrent program produces race conditions, making the results unpredictable and self-checking impossible. That is why early approaches prevented collisions to avoid unpredictable outcomes (Sullivan et al., 1995).

Since collisions are key to exposing bugs, recent approaches relied on value prediction to enable race-containing self-checking tests (Adir; Shurek, 2002).

The limitations of race-free self-checking tests and the complexity of predicting values for race-containing self-checking tests can both be avoided when verification relies on an external MCM checker (Saha et al., 1995; Hangal et al., 2004; Manovit; Hangal, 2006; Roy et al., 2006; Shacham et al., 2008; Hu et al., 2012; Graf; Andrade; Santos, 2023), because generators can synthesize *tests without results*. This largely simplifies test generation, and yet leads to effective tests with intensive collisions.

Quite distinct test representations have been used by DTG approaches reported in the literature. A general representation based on disjoint directed acyclic graphs (Corno;

Cumani; Squillero, 2003) was used to build the McVersi Test Generator (Elver; Nagarajan, 2016), which relies on Genetic Programming. A representation based on a constraint network (Adir et al., 2004) was used for building a generator based on Bayesian Networks (Fine; Ziv, 2003).

To assess the proper choice of test representations, it should be noted that the functional validation of a multicore chip or a multichip multiprocessor at design time usually follows a hierarchical methodology. First, we validate the components of a core, then the individual core as a whole, and finally the interaction among cores. The first two phases usually address issues (e.g. pipeline, dynamic issue and scheduling) that are quite different from those handled by the third phase (e.g. non-determinism, cache coherence, memory consistency). For this reason, it is convenient to use tools adapted to those different verification domains, which may rely on distinct test representations. Therefore, despite the conceptual advantage of having a unified test representation for all validation phases, too general representations cannot directly encode shared-memory behaviors, as desirable for defining effective actions for an agent.

Specific test representations targeting shared-memory behavior have been reported. Generators based on an Euler tour mechanism (Qin; Mishra, 2012; Lyu et al., 2019) cast the entire state space of a coherence protocol into a state machine graph. Their underlying test representation is a path in that graph. Albeit adequate for inducing coherence behavior for a given shared location, such path-based representation is not suitable for inducing behaviors involving distinct shared locations (as required by MCMs). On the other hand, test representations adopted for RTG (Hangal et al., 2004; Manovit; Hangal, 2006) and for DTG (Andrade et al., 2018; Andrade et al., 2020; Pfeifer et al., 2020) often consist of tuples of parameters, used as directives for commanding an RTG engine. However, representations based on RTG directives have an inherent drawback: the same directive may randomly induce different tests leading to quite distinct coverage (Fine; Ziv, 2003).

To avoid that randomness might impair learning (thereby limiting coverage), the conversion of a test encoding into an actual test program should be *deterministic*, and each encoding should produce a *unique* test.

4.3 THE PROPOSED TEST REPRESENTATION

This section first discusses requirements for an effective test representation when targeting shared-memory verification. Then it proposes a test encoding leading to a canonical representation. Finally, it shows an illustrative example.

4.3.1 Pragmatic Requirements

A suitable encoding should induce memory events captured by MCMs and coherence protocols, as follows.

4.3.1.1 *Deliberate Lack of Synchronization*

Synchronization primitives impose orderings between operations from distinct threads. Since they are intended to avoid data races, they end up preventing collisions of memory operations from distinct threads on the *same location*. However, collisions are crucial to foster coverage and error discovery (Adir; Shurek, 2002; Shacham et al., 2008; Elver; Nagarajan, 2016). Besides, synchronization may also impose orderings between operations from distinct threads involving *different locations*. Let us assume, for instance, that a design contains a consistency error: it incorrectly admits the relaxation of the order between two operations involving different locations in a given thread. Due to synchronization and depending on the execution instance of the concurrent program, this wrong ordering might not be observed by another thread, making the consistency error harder to detect. Therefore, to expose errors faster and avoid limiting coverage, a test program intended for DTG should not contain synchronization primitives¹.

4.3.1.2 *Ability to Induce Permission Requests*

A coherence protocol defines transactions that are specified by a finite state machine (FSM). A permission request is a coherence transaction that occurs, for instance, when a core asks permission to write to a local copy of a shared location (when executing a store operation), and another core contains a copy of the same location in its private cache, as a result of a previous operation (either a load or a store). Therefore, permission requests leading to core interactions may be induced when two operations from *different* threads involve the *same* location. Such requests induce state transitions in the FSMs of private cache controllers and in the FSM of the directory controller.

4.3.1.3 *Ability to Induce Eviction Requests*

An eviction request is a coherence transaction that occurs when two operations involve *different* locations mapping to the *same* cache line, and the replacement criterion (e.g. Least Recently Used (LRU)) happens to choose the same cache block that is currently holding a copy of the first memory location for holding a copy of the second one. To make

¹ Litmus tests (Nagarajan et al., 2020) are the proper means of handling synchronization primitives for MCM verification at the expense of limited coverage (Elver; Nagarajan, 2016). That is why litmus tests and directed tests are *complementary* validation approaches.

space for the second one, the first block must be evicted from the cache, which may require a write-back action to memory when the evicted block is dirty. In a private cache, eviction requests may be induced only when those two operations belong to the *same* thread. In a shared cache, such as the last-level cache, replacement requests may also be induced when those operations belong to *different* threads. Therefore, eviction requests may induce protocol state transitions in the FSMs of both private or shared cache controllers.

Thus, a test encoding tailored to the verification of coherent shared-memory behavior should be able to induce load and store operation *types* in a non-synchronized program with multiple *threads*, involving possibly different *shared locations* that might be mapped to the same or to distinct *cache lines*.

4.3.2 Theoretical Requirements

From the pragmatic requirements discussed above, let us derive abstract requirements to provide representation guarantees. Let e denote a test encoding, and let $\mathbb{E}$ be the set of all encodings. Every encoding $e \in \mathbb{E}$ should be a *unique* representation of a test and every possible test must be representable by some $e \in \mathbb{E}$. This ensures that the encoding space $\mathbb{E}$ is *not limited* by the adopted representation, although such space should be *constrained* for bounding test generation.

To formally establish the requirements of a proper representation, let us define *necessary* conditions for permission and eviction requests, as follows.

Definition 4.1 (Collision Relation). Two memory operations collide iff they reference the same shared location.

Definition 4.2 (Competition Relation). Two memory operations compete for the same cache line iff they reference locations whose effective addresses have the same index field.

Finally, for the sake of completeness, let us formalize relations capturing the well-known notions of thread and type.

Definition 4.3 (Thread Relation). Two memory operations lie in the same thread iff the fetching order of their respective instructions is specified by program order.

Definition 4.4 (Type Relation). Two memory operations have the same type iff they perform the same memory behavior.

4.3.3 The Canonical Encoding

A suitable representation for a multithreaded test program able to foster permission and eviction requests should rely on the relations above. The proposed test representation relies on them to encode a test, as follows.

Definition 4.5 (Test encoding). Let $O = \{o_1, o_2, \dots, o_n\}$ be the set of memory operations. Let s, k, p , and m be the numbers of distinct shared locations, cache indices, threads, and operation types, respectively. A test encoding e consists of the following collections of subsets of O :

- $\mathcal{O}^{co} = \{\bar{O}_i^{co}\}$, with $i = 1, 2, \dots, s$, where $\bar{O}_i^{co}$ is a subset of colliding operations;
- $\mathcal{O}^{sh} = \{\bar{O}_i^{sh}\}$, with $i = 1, 2, \dots, k$, where $\bar{O}_i^{sh}$ is a subset of competing operations;
- $\mathcal{O}^{th} = \{\bar{O}_i^{th}\}$, with $i = 1, 2, \dots, p$, where $\bar{O}_i^{th}$ is a subset of operations forming a thread.
- $\mathcal{O}^{ty} = \{\bar{O}_i^{ty}\}$, with $i = 1, 2, \dots, m$, where $\bar{O}_i^{ty}$ is a subset of operations of same type.

such that $o_x, o_y \in \bar{O}_i^{co} \Rightarrow o_x, o_y \in \bar{O}_j^{sh}$.

Note that, to distinguish the collections, we employ different identifiers, which we name as follows: *co* for color, *sh* for shade, *th* for thread, and *ty* for type. Table 6 helps the understanding of the correspondence between symbols and their meanings. The final constraint in Definition 4.5 enforces valid addresses: all operations referencing the same location, must be associated with the same index field for ensuring that a single effective address will be assigned to that location at linking time.

Table 6 – Meaning of the symbols used to define the canonical representation

Number of		Collection identifier		Set of operations	
s	shared locations	co	color	$\bar{O}_i^{co}$	referencing same location i (out of s locations)
k	cache indices	sh	shade	$\bar{O}_i^{sh}$	referencing same cache line i (out of k lines)
p	threads	th	thread	$\bar{O}_i^{th}$	forming same thread i (out of p threads)
m	types	ty	type	$\bar{O}_i^{ty}$	of same type i (out of m types)

The number of distinct types of memory operations (m) is fixed for a given *instruction set architecture* (ISA) and the number of threads (p) is fixed for a given microarchitecture. However, the amount of operations (n), locations (s), and cache indices (k) can vary from one test to another.

4.3.4 Theoretical Guarantees

Let $E(n, s, k, p, m)$ denote the set of all encodings satisfying Definition 4.5 for a given choice of n, s, k, p , and m .

Theorem 4.1 (Uniqueness). Every $e \in E(n, s, k, p, m)$ is unique.

Proof: The relations in Definitions 4.1, 4.2, 4.3, and 4.4 are all reflexive, symmetric and transitive. Since they are equivalence relations, each collection $\bar{O}^{co}, \bar{O}^{sh}, \bar{O}^{th}$, and $\bar{O}^{ty}$ is a partition of O (Cormen et al., 2009). Since each partition of a set is unique (because it consists of non-empty *disjoint* subsets), every encoding e is unique. ■

Theorem 4.2 (Completeness). Any arbitrary collections $\overset{co}{\mathcal{O}}, \overset{sh}{\mathcal{O}}, \overset{th}{\mathcal{O}}$, and $\overset{ty}{\mathcal{O}}$ can be represented by some $e \in E(n, s, k, p, m)$.

Proof: Since the relations in Definitions 4.1, 4.2, 4.3, and 4.4 are equivalence relations, and therefore the collections $\overset{co}{\mathcal{O}}, \overset{sh}{\mathcal{O}}, \overset{th}{\mathcal{O}}, \overset{ty}{\mathcal{O}}$ are partitions of $\mathcal{O}$ (Cormen et al., 2009), any arbitrary variants of them can be built by moving elements between disjoint subsets, i.e. between $\overset{co}{O}_i$ and $\overset{co}{O}_{j \neq i}$, $\overset{sh}{O}_i$ and $\overset{sh}{O}_{j \neq i}$, $\overset{th}{O}_i$ and $\overset{th}{O}_{j \neq i}$, and between $\overset{ty}{O}_i$ and $\overset{ty}{O}_{j \neq i}$. Thus, there exists some encoding $e \in E(n, s, k, p, m)$ that represents each arbitrary variant. ■

4.3.5 Implementation and Example

In our implementation of the canonical encoding, the partition induced by the threads is explicitly represented by the disjoint subsets $\overset{th}{O}_i$, which are kept as ordered lists (for convenience of generation) so as to capture program order. The other partitions are implicitly represented by means of operation attributes. The attribute *color* denotes the shared *location* referenced by a given operation, and the attribute *shade* denotes the *index field* of the effective address to be assigned to the referenced location. The attribute *type* denotes an operation as a load (L) or a store (S)². As a result, each distinct color i serves as identifier for a different subset $\overset{co}{O}_i$, each distinct shade i serves as identifier for a different subset $\overset{sh}{O}_i$, and L and S serve as identifiers for distinct types.

Let us consider an example of encoding for a test program with two threads: $e = \{e_1, e_2\}$, where e_1 and e_2 are disjoint subsets of the memory operations corresponding to threads 1 and 2, respectively. Let X be a symbol for the type identifier, σ be a shade identifier, and c be a color identifier. Let the decorated symbol $\overset{\sigma}{X}_c$ represent a memory operation.

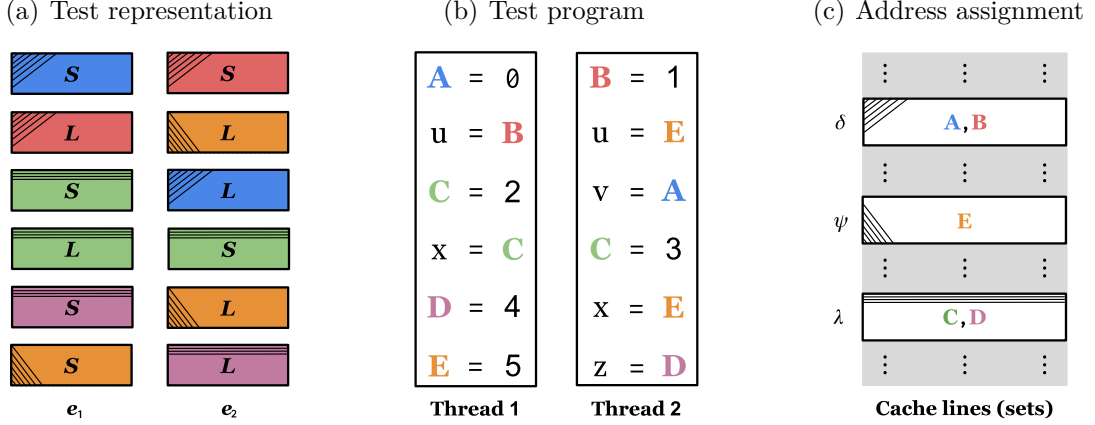
A thread is represented as a sequence of decorated symbols. Suppose that each thread is a sequence of six operations, involving three shades (δ, ψ and λ) and five colors: blue ($\textcolor{blue}{b}$), red ($\textcolor{red}{r}$), green ($\textcolor{green}{g}$), magenta ($\textcolor{magenta}{m}$), and orange ($\textcolor{orange}{o}$). An example of a test encoding for a dual-thread program is:

$$e = \{e_1, e_2\} = \{(\overset{\delta}{S}_{\textcolor{blue}{b}}, \overset{\delta}{L}_{\textcolor{red}{r}}, \overset{\lambda}{S}_{\textcolor{green}{g}}, \overset{\lambda}{L}_{\textcolor{green}{g}}, \overset{\lambda}{S}_{\textcolor{magenta}{m}}, \overset{\psi}{S}_{\textcolor{orange}{o}}), (\overset{\delta}{S}_{\textcolor{red}{r}}, \overset{\psi}{L}_{\textcolor{orange}{o}}, \overset{\delta}{L}_{\textcolor{blue}{b}}, \overset{\lambda}{S}_{\textcolor{green}{g}}, \overset{\psi}{L}_{\textcolor{orange}{o}}, \overset{\lambda}{L}_{\textcolor{magenta}{m}})\}$$

Operations with the same color represent collisions on the same shared location, and operations with same shade represent competition for the same cache line. Fig. 10 shows the synthesis of a test program from the encoding above.

Fig. 10(a) shows a schematic view of the encoding, where a box represents an operation consisting of a symbol decorated with color and shade. Fig. 10(b) shows the

² For simplicity, we restrict our implementation to load and store archetypes. This complies with modern instruction-set architectures (such as ARMv8, RISC-V, and IBM Power), where reserved load and store instructions replace old-style atomic exchange instructions.

Figure 10 – Synthesis of a test program from an encoding $e = \{e_1, e_2\}$.

resulting source program, where capital letters denote shared memory locations and small letters represent variables allocated in registers. Shades cannot be encoded in the source program, because they are abstractions for index fields of effective addresses, which are produced after compiling and linking the source program. To schematically represent them, Fig. 10(c) indicates the distinct lines (of the shared last-level cache) to which are mapped the locations with same shade. Distinct shades are implemented as different index fields by means of proper address assignment, just before the linker produces the executable code.

4.4 TEST GENERATION AS AN OPTIMIZATION PROBLEM

This section formalizes the notions required to the formulation of the target problem.

Let n_{min}, n_{max} and s_{min}, s_{max} be user-defined *range constraints* for the amount of operations (n) and locations (s).

Definition 4.6. The allowed numbers of memory operations are $N = \{n : n_{min} \leq n \leq n_{max} \text{ and } n = 2^i \text{ with } i \in \mathbb{N}\}$.³

Definition 4.7. The allowed numbers of shared locations are $S = \{s : s_{min} \leq s \leq s_{max} \text{ and } s = 2^i \text{ with } i \in \mathbb{N}\}$.

Let k denote the number of distinct cache indices available for assigning effective addresses to the s shared locations of a given test program. Without loss of generality, we constrain the values that can be assigned to k such that exactly s/k locations compete for each cache line⁴, as follows.

³ Although the allowed values could be induced by the range of some generic function $f(i)$, without loss of generality, we assume $f(i) = 2^i$.

⁴ The choice of such uniform distribution maximizes the probability of controlling replacements in cache sets for a given setting of s .

Definition 4.8. For a given s , the allowed numbers of cache indices are $\kappa(s) = \{k : 1 \leq k \leq s \text{ and } s \bmod k = 0\}$.

Let p be the number of threads, and let τ_i be the number of operations in thread i . Thread sizes are constrained as follows.

Definition 4.9. For a test program with p threads, their sizes are such that $\tau_i \geq 1$ and $\sum_{i=1}^p \tau_i = n$.

Definition 4.10. The *generation space* induced by the range constraints is $\mathbb{G} = \{(n, s, k) : (n, s) \in \mathbb{N} \times \mathbb{S} \text{ and } k \in \kappa(s)\}$.⁵

Let e be an encoding serving as a representation of a test program, and let $E(n, s, k)$ be the set of encodings for a given choice of n , s , and k .

Definition 4.11. The *encoding space* induced by the generation space is $\mathbb{E} = \{e : e \in E(n, s, k) \text{ and } (n, s, k) \in \mathbb{G}\}$.

Let T be a predefined time limit and let E be a set of encodings. Let $cv(E)$ denote the cumulative coverage obtained with the test programs induced by E , and let $t(E)$ denote their cumulative runtime. We cast test generation as an optimization problem for which the objective is to maximize coverage within the predefined time limit, as follows.

Problem 1. Given the encoding space $\mathbb{E}$, find a subspace $E_o \subseteq \mathbb{E}$ with $t(E_o) \leq T$ such that $cv(E_o) \geq cv(E)$ for all $E \subseteq \mathbb{E}$ with $t(E) \leq T$.

4.5 THE LEARNING-BASED GENERATOR

To solve Problem 1, we have built a Learning-Based Generator (LBG) directed by a Reinforcement Learning agent, and we have designed its actions with basis on our encoding.

Fig. 11 shows the components of the proposed DTG approach. It consists of a Directing Engine, which produces an encoding (e) serving as a test representation. The encoding is converted by a Test Generation Engine into an actual test program. The execution of the test is simulated for some *design under verification* (DUV). A Coverage Engine evaluates its contribution to the cumulative coverage (cv), which is fed back to the Directing Engine. Based on the coverage improvement, the Directing Engine iteratively generates new encodings until the coverage goals are met, a time limit is reached, or a design error is found.⁶

⁵ Without loss of generality, we assume a fixed number of threads (p) and a fixed number of types (m), as determined by a given design under verification.

⁶ For simplicity, we omit from Fig. 11 the MCM checker, which is tied to the DUV, and is able to detect design errors.

Figure 11 – A verification framework for an encoding-based DTG approach

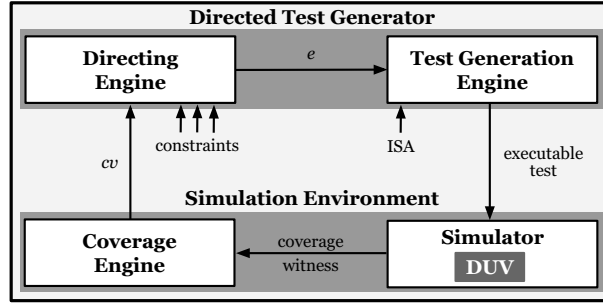


Fig. 12 casts the DTG approach in Fig. 11 into a Reinforcement Learning formulation, where an Agent takes an action on the environment, observes its reward, and defines the next action. Depending on the coverage reward (rw), the Agent defines the next action (a) to the Encoder. For the target number of threads (p), and a given point of the generation space (g), the Encoder accomplishes that action to produce a new test representation (e).

Figure 12 – Anatomy of the Directing Engine for the proposed approach

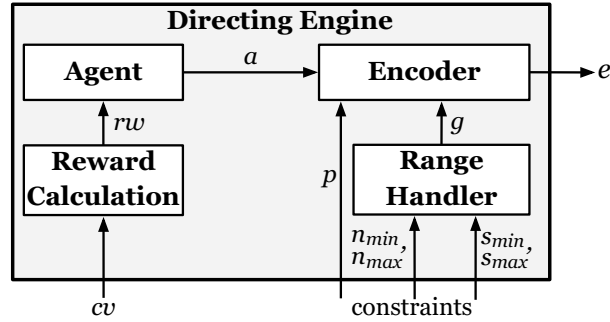
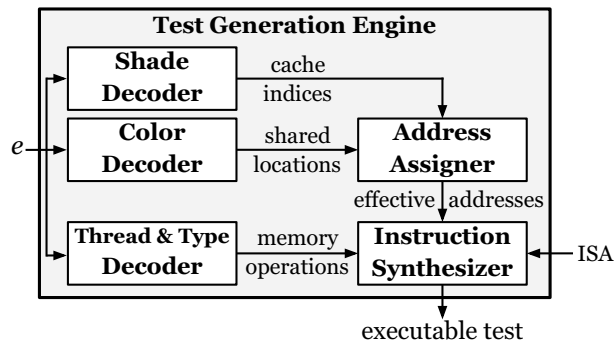


Fig. 13 shows that the Test Generation Engine decodes the four components of our test representation. Shade and color identifiers are used to assign index and tag to each shared location, forming a binary address. The executable test is obtained by compiling operations into instructions, and linking code and data according to the induced address assignment.

Figure 13 – Anatomy of the Test Generation Engine for the proposed approach



4.5.1 Design of Actions based on the Canonical Representation

The proposed representation provides guarantees for defining proper actions, because it is canonical. Uniqueness (Theorem 4.1) prevents actions from inducing redundant tests and completeness (Theorem 4.2) prevents actions from limiting the test suite. The design of actions is inherently domain-specific, and action representation can strongly affect performance. For those reasons, our action design was inspired by a legacy of *constrained* RTG: the alternation of relaxation and enforcement of constraints as key to controlling proper DUV stimulation.

Actions are taken over points of the generation space $\mathbb{G}$, which correspond to distinct generation parameters. When the agent takes a *relaxing* action, it allows generation parameters to be relaxed when inducing a test variant, but when it takes a *constraining* action, it enforces generation parameters while inducing a test variant. It should be noted that our approach fully eliminates the inconvenience of trying to direct a *random* test generator, but it preserves the important role of *constraints*. Indeed, we let actions manage constraints for directing a *deterministic* test generator.

In the context of test generation, the ultimate goal of an action is to provoke some change in the last synthesized program to give rise to the next program for the test suite. For instance, an action could be as simple as turning a single load into a store in a single thread, or as complex as exchanging the cache line addresses of two distinct locations in all threads. Since we deal with test representations, an action can be seen as a transformation of one encoding into another. However, the number of possible transformations is huge. To cope with the huge number of low-level actions, we group similar transformations into a single action category that has the power to induce any arbitrary action of a same kind. Each such a category can be seen as an *abstract action*. Without loss of generality, this chapter focuses on three abstract actions (categories), as follows.

1. **new-point**: create some arbitrary encoding induced by a new point $g \in \mathbb{G}$.
2. **exchange-shades**: given an encoding, exchange the shades of a pair of colors in all threads.
3. **mutate-type**: given an encoding, modify the type of a single operation in a single thread.

Category 1 induces relaxing actions, because it produces an encoding that has new values for (n, s, k) . Such point may lie on uncharted territory of the generation space. Categories 2 and 3 induce constraining actions, because they produce encodings that preserve the values for (n, s, k) , i.e. they recode a previous encoding induced by the same point, which certainly lies on charted territory of the generation space.

Category 2 largely changes shared-memory behavior, because (under true sharing) the new shading is likely to modify eviction patterns in all threads referencing the

involved locations.

At first glance, Category 3 might seem too punctual to be effective. To explain why this is not the case, let us analyze the conditions to foster permission requests resulting in interactions between *two or more* processors. This happens when: (1) a load misses in a local cache and has to check if the shared location was updated by a *store* in a remote cache, and (2) a *store* misses in a local cache and has to check if there is a stale copy of the shared location that was brought to a remote cache by either a load or a store.⁷ Both cases involve a store, which motivates the following relation.

Definition 4.12. (*Conflict Relation*) Two memory operations conflict if they collide and at least one of them is a store.⁸

Since Category 3 may globally change conflict patterns of *many* operations colliding at a same location in distinct threads, it may considerably change shared-memory behavior.

4.5.2 Encoder

This section shows how the Encoder performs the Agent’s actions. Algorithm 1 summarizes the action-sensitive encoder. The routines ENCODE and RECODE handle relaxing and constraining actions, respectively. We let *last_e* denote a static global variable that keeps the last generated encoding.

Algorithm 1 ENCODE(g, p, a)

```

1: if  $a = \text{new-point}$  then
2:    $e \leftarrow \text{ENCODE}(g, p)$ 
3: else
4:    $e \leftarrow \text{RECODE}(\text{last\_e}, g, p, a)$ 
5: end if
6:  $\text{last\_e} \leftarrow e$ 
7: return  $e$ 

```

Algorithm 2 implements some relaxing action. Let τ_i be the number of operations of a thread i , and let $e_i[x]$ be its x -th operation slot. Thread, type, and color are arbitrarily chosen for each operation, but not shade. As required by Definition 4.5, the routine PROPER-SHADE constrains all operations of a given color to have the same shade.

Algorithm 3 implements constraining actions. In the first case (lines 3-8), some action from Category 2 is implemented: some color is arbitrarily chosen, then the routine

⁷ In protocol jargon, the first case induces GetS and Fwd-GetS transactions, and the second case induces a GetM transaction, and either a Fwd-GetM transaction or possibly multiple Invalidate transactions (Nagarajan et al., 2020).

⁸ The conflict relation is symmetric, but it is neither reflexive nor transitive. Since it is not an equivalence relation, it cannot provide representation guarantees, but it can serve as one of the keys to effective actions.

Algorithm 2 ENCODE(g, p)

```

1:  $(n, s, k) \leftarrow g$ 
2: for  $i = 1, 2, \dots, p$  do
3:    $\tau_i \leftarrow 0$ 
4: end for
5: for  $j = 1, 2, \dots, n$  do
6:    $i \leftarrow \text{RANDOM}(\{1, 2, \dots, p\})$ 
7:    $\tau_i \leftarrow \tau_i + 1$ 
8:    $e_i[\tau_i].\text{type} \leftarrow \text{RANDOM}(\{L, S\})$ 
9:    $e_i[\tau_i].\text{color} \leftarrow \text{RANDOM}(\{1, 2, \dots, s\})$ 
10:   $e_i[\tau_i].\text{shade} \leftarrow \text{PROPER-SHADE}(k, e_i[\tau_i].\text{color})$ 
11: end for
12: return  $\{e_1, e_2, \dots, e_p\}$ 

```

SELECT-SUITABLE-COLOR returns some color y with a different shade, and finally the routine EXCHANGE-SHADES returns an encoding where the shades originally associated with colors y and c are exchanged. In the second case (lines 9-16), some action from Category 3 is implemented: a thread and one of its slots are arbitrarily selected, then the type of the operation in that slot is toggled.

Algorithm 3 RECODE(e, g, p, a)

```

1:  $(n, s, k) \leftarrow g$ 
2: switch  $a$  do
3:   case exchange-shades
4:      $c \leftarrow \text{RANDOM}(\{1, 2, \dots, s\})$ 
5:     if  $k > 1$  then
6:        $y \leftarrow \text{SELECT-SUITABLE-COLOR}(c, s)$ 
7:        $e \leftarrow \text{EXCHANGE-SHADES}(e, y, c)$ 
8:     end if
9:   case mutate-type
10:     $i \leftarrow \text{RANDOM}(\{1, 2, \dots, p\})$ 
11:     $x \leftarrow \text{RANDOM}(\{1, 2, \dots, \tau_i\})$ 
12:    if  $e_i[x].\text{type} = L$  then
13:       $e_i[x].\text{type} \leftarrow S$ 
14:    else
15:       $e_i[x].\text{type} \leftarrow L$ 
16:    end if
17: return  $e$ 

```

4.5.3 Directing Engine

Algorithm 4 delimits the generation space according to Definition 4.10 (line 2), and visits its points in arbitrary order (lines 3, 12). Initially, it takes a relaxing action (line 4), commands the Encoder to obtain a first encoding (line 5), and then invokes the Test Generation Engine (line 6) to define the initial coverage, allowing the initialization of the reward value (line 8). Finally, the algorithm iteratively invokes the Agent, the

Encoder, and the Test Generation Engine until full coverage is attained or the predefined time limit is reached (lines 9-18).

Algorithm 4 DIRECTING ENGINE($n_{\min}, n_{\max}, s_{\min}, s_{\max}, p$)

```

1:  $E \leftarrow \emptyset$ 
2:  $\mathbb{G} \leftarrow \text{GENERATION SPACE}(n_{\min}, n_{\max}, s_{\min}, s_{\max})$ 
3:  $g \leftarrow \text{RANDOM}(\mathbb{G})$ 
4:  $a \leftarrow \text{new-point}$ 
5:  $e \leftarrow \text{ENCODER}(g, p, a)$ 
6:  $\text{TGE}(e)$ 
7:  $E \leftarrow E \cup \{e\}$ 
8:  $rw \leftarrow cv(E)$ 
9: repeat
10:    $a \leftarrow \text{AGENT}(rw)$ 
11:   if  $a = \text{new-point}$  then
12:      $g \leftarrow \text{RANDOM}(\mathbb{G})$ 
13:   end if
14:    $e \leftarrow \text{ENCODER}(g, p, a)$ 
15:    $\text{TGE}(e)$ 
16:    $E \leftarrow E \cup \{e\}$ 
17:    $rw \leftarrow cv(E) - cv(E \setminus \{e\})$ 
18: until  $cv(E) = 1 \vee t(E) > T$ 

```

To improve readability, we have omitted from the above algorithms the snippets that preserve properties of the partitions and provide compliance with the constraints, because we had already formalized them in Sections 4.3 and 4.4.

4.6 EXPERIMENTAL EVALUATION

We compared the LBG with three directed test generators targeting shared-memory verification. The McVersi Test Generator (MTG) (Elver; Nagarajan, 2016) relies on a *general*-purpose encoding. The Hybrid Test Generator (HTG) (Andrade et al., 2020) and the Reinforcement Learning Generator (RLG) (Pfeifer et al., 2020) are both based on the same domain-*specific* encoding.

For each generator, we evaluated the cumulative coverage reached for distinct time limits. The experiments targeted 32-core designs with distinct coherence protocols (3-level MESI and 2-level MOESI).

We compared the generators under two metrics: structural transition coverage (as defined in (Elver; Nagarajan, 2016)) and functional transition coverage (as defined in (Andrade et al., 2020)). While running a test, the number of distinct transitions covered in the code of each cache controller’s FSM was tracked. The functional metric distinguishes transitions from *identical* controller instances in different core domains. For the structural metric, no distinction is made between them.

4.6.1 Infrastructure and Setup

We relied on gem5’s infrastructure (Binkert et al., 2011) for simulation and design representation (we choose *O3*, *Ruby*, and *simple* for CPU, memory, and network models, respectively). We have built two different 32-core designs. Each design adopted a distinct directory-based coherence protocol from gem5. One design relied on a 2-level (L1, L2) MOESI protocol, another on a 3-level (L0, L1, L2) MESI protocol. For the cache hierarchy, we adopted 4KiB (directed-mapped) private caches at L0, 64KiB (2-way) private caches at L1, and a 2MiB (8-way) shared L2 cache, all with same block size (64 bytes).

We performed experiments with all generators under the same range constraints: $n_{min} = 1\text{Ki}$ and $n_{max} = 64\text{Ki}$ for the number of operations, $s_{min} = 4$ and $s_{max} = 128$ for the number of locations. However, the allowed values within those ranges had to comply with generator’s differences, as follows.

LBG, RLG, and HTG can generate programs with *variable* numbers of operations in the same test suite. However, in a test suite generated by MTG, all programs must have the same *fixed* number of operations. In other words, MTG solves an instance of Problem 1 that is more restrictive than LBG’s. To mimic MTG’s target problem instance, we had to constrain the range of operations of LBG (such that $n_{min} = n_{max}$) for performing experiments with suites of fixed-size tests. That is why we also report LBG results for suites with fixed-size tests (1Ki and 64Ki operations), so as to compare them with MTG when solving its more restrictive problem instance.

LBG, RLG, and HTG dynamically exploit *multiple* address-space constraints (as a result of varying the number of shades in LBG or varying the parameter k in RLG and HTG). However, the MTG requires the static definition of a *single* address-space constraint, which is called *test memory size* (TM). To accommodate such difference, we report MTG results under the (best) static constraint defined in (Elver; Nagarajan, 2016), i.e. $\text{TM} = 8\text{KiB}$.⁹

MTG is free to select as many locations as available within its useful address space. For LBG, RLG, and HTG, the number of locations is constrained by Definition 4.8. To accommodate such difference, we compared the generators under the same upper bound on the number of useful locations (Elver; Nagarajan, 2016). Under Definition 4.8, the maximum number of block-aligned locations for $\text{TM} = 8\text{KiB}$ is 128, making $s_{max} = 128$ a proper choice.

The Agent employed by RLG is an adaptation of the Rainbow agent (Hessel et al., 2018), where the original deep neural network was replaced by a recurrent neural network (Pfeifer et al., 2020). As the tests used for training would not impair coverage, but actually contribute to its cumulative effect, the nature of the problem allowed RLG to opt for online training. At the start of every test suite execution, RLG used a new set

⁹ The MTG code is available in the public domain (Elver, 2016). We preserved all genetic parameters exactly as they were originally set in (Elver; Nagarajan, 2016).

of random weights for the network, and trained them during its execution. Thus, for a proper comparison, we adopted for LBG the same agent (except for our distinct actions) and the same training policy.

Each generated test was executed five times under different simulation states (not related to the test itself) in such a way that the distinct executions of the same test are all perturbed differently (Elver; Nagarajan, 2016). To get the reported values, we launched each generator five times with different seeds, and we took the median values of the resulting distributions.

Runtimes were measured on a Linux server (Intel Xeon E5-2640 v4, 2.4 GHz, with 128GB of main memory).

4.6.2 Action Effectiveness under the Proposed Approach

This section reports the evaluation of action effectiveness under the proposed representation. Let us consider ablations of LBG with incrementally larger sets of actions: $A1 = \{\text{new-point}\}$ induces only relaxing actions, $A2 = \{\text{new-point}, \text{exchange-shades}\}$ and $A3 = \{\text{new-point}, \text{exchange-shades}, \text{mutate-type}\}$ induce constraining actions too. Fig. 14 shows their coverage evolution under the functional metric when targeting the 3-level MESI DUV with distinct action sets.

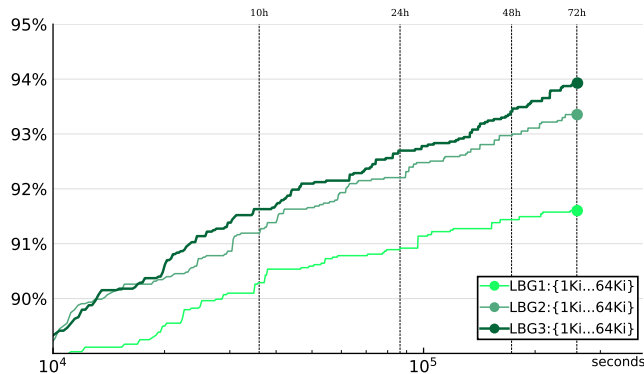


Figure 14 – Impact of distinct sets of actions on coverage evolution

Fig. 14 shows that, when an agent takes only relaxing actions (LBG1), it induces smaller coverage as compared with agents that combine both relaxing and constraining actions (LBG2 and LBG3). In particular, note that **exchange-shades** largely impacts coverage, whereas **mutate-type** has smaller impact. However, **mutate-type** is essential for LBG3 to reach larger coverage than LBG2.

The results obtained with our minimalist agent indicate that *our representation enables the definition of effective actions*. This allows us to envisage the definition of a larger set of action categories to pave the way towards the future design of sophisticated agents for DTG. However, substantial impact on DTG can already be obtained with our minimalist agent, as reported in the next sections. All the following experiments will evaluate the LBG, as compared to related work, when the underlying agent uses the largest action set (i.e. $A3$).

4.6.3 Impact on Coverage for a Given Time Limit

Table 7 – Median structural coverage reached for a given time limit

32-core DUV	T	{1Ki, ... , 64Ki}			{1Ki}		{64Ki}	
		HTG	RLG	LBG	LBG	MTG	LBG	MTG
2-level MOESI	24 h	95.25%	95.25%	97.44%	95.35%	91.61%	96.72%	95.62%
	48 h	95.25%	95.25%	97.81%	95.81%	91.61%	97.45%	96.35%
	72 h	95.25%	95.25%	97.81%	95.81%	91.61%	97.81%	96.35%
3-level MESI	24 h	96.28%	96.28%	97.21%	95.99%	95.35%	96.74%	95.81%
	48 h	96.28%	96.28%	97.21%	96.35%	95.81%	97.21%	96.28%
	72 h	96.28%	96.28%	97.21%	96.35%	95.81%	97.21%	96.28%

Table 7 shows the *structural coverage* obtained with suites of variable and fixed-size tests. It captures the protocol complexity, regardless of core count. The slight variation in coverage with largely growing deadlines reveals events that are hard to stimulate. Thus, that variation can be used to compare the ability of the generators to stimulate ultra-unlikely events.

The generators relying on a test representation that encodes directives for an RTG engine (RLG and HTG) led to inferior coverage for both DUVs. Thus, they seem unable to handle ultra-unlikely events. In particular, since RLG and LBG rely on the same agent structure (except for their different actions), but on distinct test representations, the superiority of LBG clearly exposes the limitation of RTG directives as test encodings.

The generators relying on deterministic test generation engines (LBG and MTG) both have the ability to stimulate ultra-unlikely events in the long run, despite their distinct test representations. However, LBG obtained the largest coverage for both DUVs and every time limit. This indicates that our problem-specific encoding of primal behaviors is more effective to stimulate rare protocol events.

Table 8 shows the *functional coverage*. It captures core interactions through hierarchical levels (e.g. from a requestor down to the directory and then up to a sharer or owner). Since coverage goals include distinct instances of the same transition in different core domains, the number of transitions to be covered is amplified by core count at every *private* level.

The limitation of generators based on RTG directives as test representations (RLG and HTG) is confirmed for the functional metric. In contrast, those based on deterministic test generation engines (LBG and MTG) were able to stimulate ultra-unlikely events captured by the functional metric, for which coverage goals are largely amplified.

When targeting the 2-level MOESI design, LBG reached the largest coverage for every time limit. In contrast, when targeting the 3-level MESI design, MTG reached the largest coverage for every time limit with the {64Ki} test suite. Although that suite happened to lead to the best solution for that specific problem instance, this is not often the case, as observed in Table 7 and on the top of Table 8.

MTG’s test representation over-constrains program generation. Its fixed-size requirement may exclude the best solution from the search space, as observed in three out of the four reported scenarios. In contrast, our canonical test representation does not limit the search space, and it accommodates flexible range constraints, which can be tightened on demand if needed.

Table 8 – Median functional coverage reached for a given time limit

32-core DUV	T	{1Ki, ... , 64Ki}			{1Ki}		{64Ki}	
		HTG	RLG	LBG	LBG	MTG	LBG	MTG
2-level MOESI	24 h	93.13%	93.13%	94.05%	90.17%	90.23%	93.98%	92.99%
	48 h	93.13%	93.13%	94.09%	93.54%	90.27%	94.01%	92.99%
	72 h	93.13%	93.13%	94.09%	93.68%	90.30%	94.12%	92.99%
3-level MESI	24 h	90.94%	90.94%	92.70%	91.06%	92.88%	92.18%	94.09%
	48 h	90.94%	90.94%	93.49%	92.01%	93.05%	93.11%	94.64%
	72 h	90.94%	90.94%	93.93%	92.64%	93.13%	93.52%	94.64%

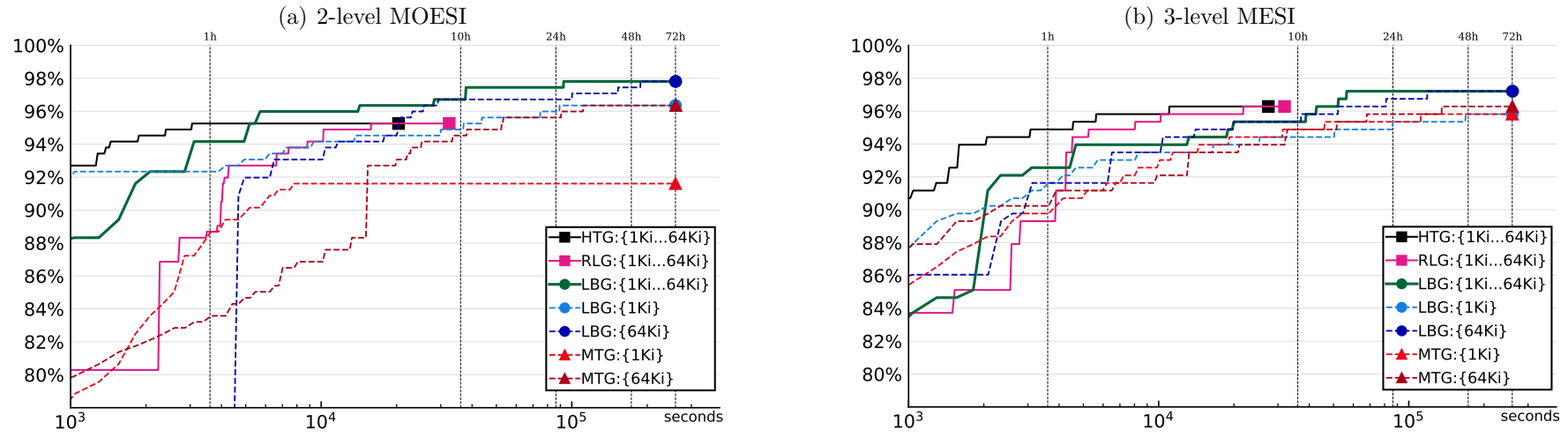
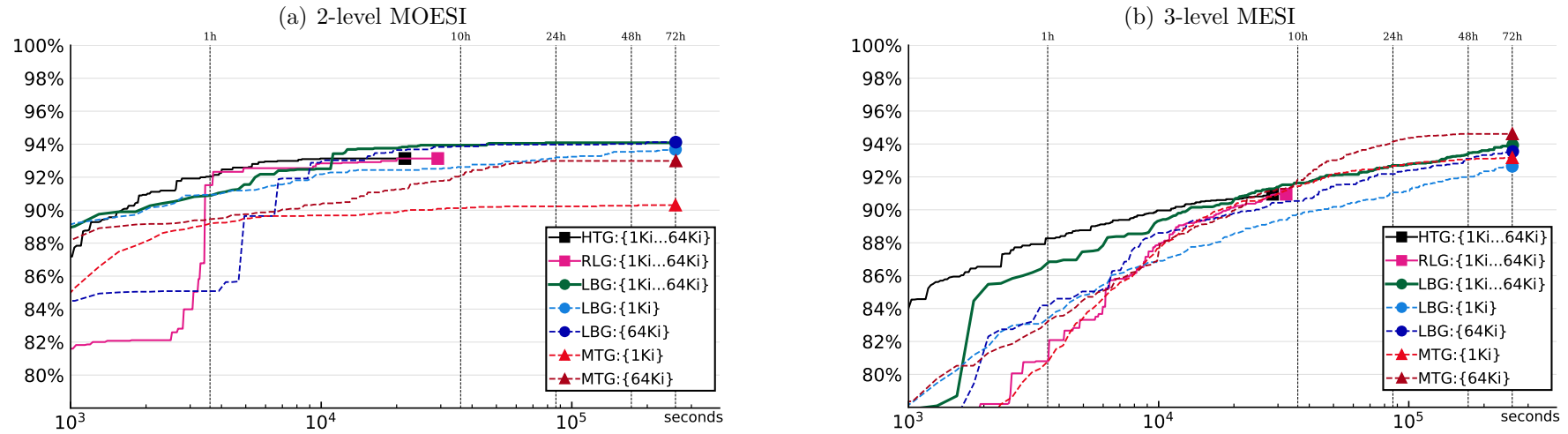
The fact that MTG’s unique mechanism (selective crossover (Elver; Nagarajan, 2016)) did not often overcome the LBG (a generator with a minimalist agent) is evidence that *our problem-specific test representation is effective*. Of course, a more elaborate learning-based generator could be built upon our encoding (with a larger set of actions and/or a deeper neural network structure for the agent), but this would *hardly* serve as further evidence of the effectiveness of our canonical representation, because results would just reflect the impact of a more powerful agent.

4.6.4 Coverage Evolution with Time

Fig. 15 shows the coverage evolution under the structural metric. Note that, most of the time, LBG has faster evolution as compared to RLG, which employs the same agent structure, but distinct actions, and a quite distinct test representation. Since LBG has half the actions of RLG, this is an evidence that our encoding grants superior exploitation during learning. LBG and MTG (64Ki) both evolve after 10 hours, but LBG is superior and reaches the largest coverage. Thus, the unique generation mechanism of MTG (selective crossover) cannot always compensate for a too general test representation.

Fig. 16 shows the coverage evolution under the functional metric. LBG is superior to RLG, confirming that our encoding improves action effectiveness with the same agent structure. Fig. 16(a) shows that, for the 2-level MOESI design, LBG is dominant with the suite of variable-size tests. In contrast, Fig. 16(b) shows that, between 10 and 72 hours, MTG is dominant for the 3-level MESI design with the {64Ki} suite. This is hardly due to MTG’s test representation (which led to inferior structural coverage for both designs and to inferior functional coverage for the 2-level MOESI design). Instead, such late dominance is probably due to its selective crossover meta-heuristic, which happened to find better solutions for that specific problem instance in that interval.

Besides, the fact that LBG has reached the largest coverage in most cases, despite the use of a minimalist agent, suggests that our test representation can leverage even faster evolution if used for the design of a larger set of actions.

Figure 15 – Median *structural* coverage evolution for 32-core designs as a function of the required timeFigure 16 – Median *functional* coverage evolution for 32-core designs as a function of the required time

4.6.5 Impact on Design Error Diagnosis

In previous sections, we reported the cumulative coverage resulting from the execution of a sequence of tests on designs containing no errors. In this section, we show the impact of our test representation on the runtime required to discover design errors. We inserted different artificial errors by changing the FSMs of the coherence protocols (either by modifying the next state or precluding some due output). Each faulty DUV contained a single, distinct error. The errors studied in our experiments are described in (Andrade et al., 2020). We selected the most challenging ones: e4 and e5 for 2-level MOESI, E4 and E5 for 3-level MESI.

For each of five distinct random seeds, we measured the runtime until the error was found and the coverage reached at the time of error discovery. Then we plotted a constellation of coverage-time points for each generator, under both metrics, and for every faulty DUV.

Fig. 17 and Fig. 18 show the constellation plots for 2-level MOESI and 3-level MESI, respectively. *All* verification episodes with LBG and MTG {64Ki} exposed *every* error. However, only error E5 was exposed by all episodes with MTG {1Ki}. Actually, 17 episodes with MTG {1Ki} failed to expose errors within the 72-hour limit: two episodes under the structural metric (Fig. 17(a)) and three episodes under the functional metric (Fig. 17(b)) failed to detect e4, one episode failed to detect E4 under each metric (Fig. 18(a) and Fig. 18(b)), and all five episodes failed to detect e5 under each metric (Fig. 17(c) and Fig. 17(d)). Since MTG {1Ki} was able to expose errors in only 57,5% of the episodes (23/40), MTG required a larger test size (64Ki) to reach 100% discovery rate. This indicates that LBG can reach discovery rates similar to MTG with smaller average test size, because it can synthesize suites where tests have variable size (from 1Ki to 64Ki). From now on, let us focus on the comparison between LBG and MTG {64Ki}.

Fig. 17 shows that LBG exposes both 2-level MOESI errors much faster than MTG. Fig. 18 shows that LBG exposes one of the 3-level MESI errors much faster than MTG, and exposes the other slower than MTG, but in this case their discovery times are within the same order of magnitude. Table 9 compares the median time for the diagnosis of every error, and shows that LBG has led to the fastest error diagnosis in 3/4 of the verification scenarios.

Note that LBG exposed each of the four errors in less than one hour (regardless of protocol type and coverage metric) for all but two verification episodes (Fig. 18(a) and Fig. 18(c)).

Table 9 – Median time for error diagnosis (in seconds)

Protocol	Error	Metric	LBG	MTG{64Ki}
2-level MOESI	e4	structural	22	4968
		functional	23	5072
	e5	structural	187	3259
		functional	199	3191
3-level MESI	E4	structural	1856	1089
		functional	1887	1413
	E5	structural	1170	20553
		functional	1158	26999

This means that the LBG time to expose errors was inferior to one hour in 95% of verification episodes (38/40). For MTG {64Ki}, the time to expose errors was inferior to one hour in only 47.5% of the verification episodes (19/40). This indicates that the use of the proposed test representation leads to the generation of high-quality tests, and it seems largely independent of protocol and coverage metric.

Recall that the functional metric distinguishes transitions from identical controller instances in distinct core domains, as opposed to the structural metric. This means that the functional metric tracks better the interaction between cores. Observe that, when the functional metric is adopted (Fig. 17(b), Fig. 17(d), Fig. 18(b), and Fig. 18(d)), LBG exposes errors with lower coverage values (on average), as compared to MTG. This indicates that, when tests are generated with the proposed test representation, errors can be exposed with lower coverage, because that representation captures properties of concurrent programs that *effectively* induce coherence transactions, which are key to the functional verification of the interaction between cores.

In short, the experimental results show that the proposed test representation tends to improve coverage evolution and error diagnosis, and this was observed for multiple verification episodes. Thus, we expect learning-based directed test generators to play a key role in Electronic Design Automation. Not only agents can learn from interaction with the simulation environment as a result of the online experience of a single verification episode, but they are likely to improve the quality of the generated test suites as the verification process evolves, as a result of the experience along successive verification episodes. That is why agents deserve proper tailoring to the target DTG problem. However, this does not seem to be the case of the agent proposed by RLG (Pfeifer et al., 2020) (and reused by LBG for a proper comparison), apparently due to a limited state representation. This motivates further research for defining a proper *state* representation for the target problem, similarly to what we have done for *action* representation in this chapter.

Figure 17 – Discovery constellations for 2-level MOESI design errors

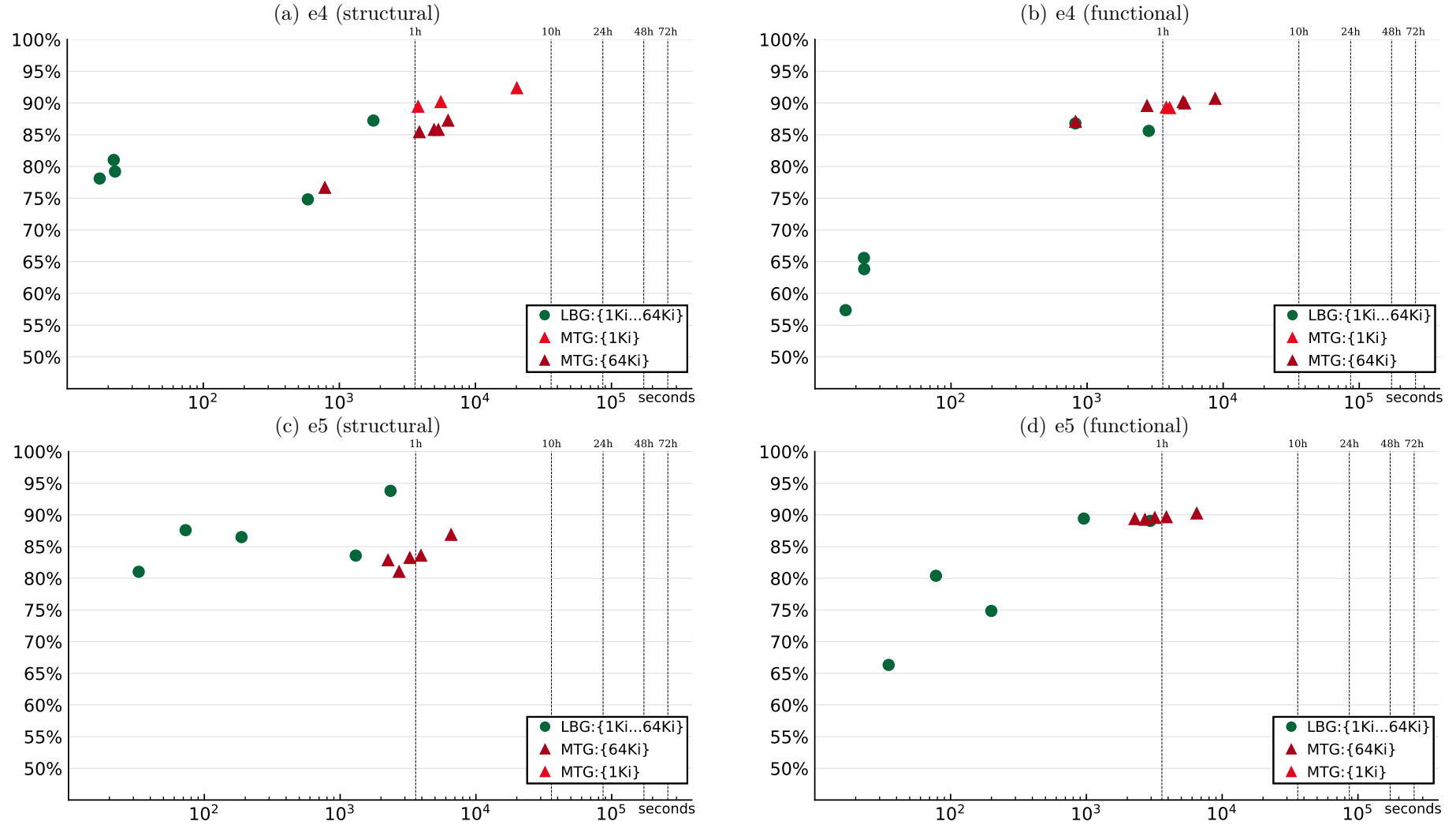
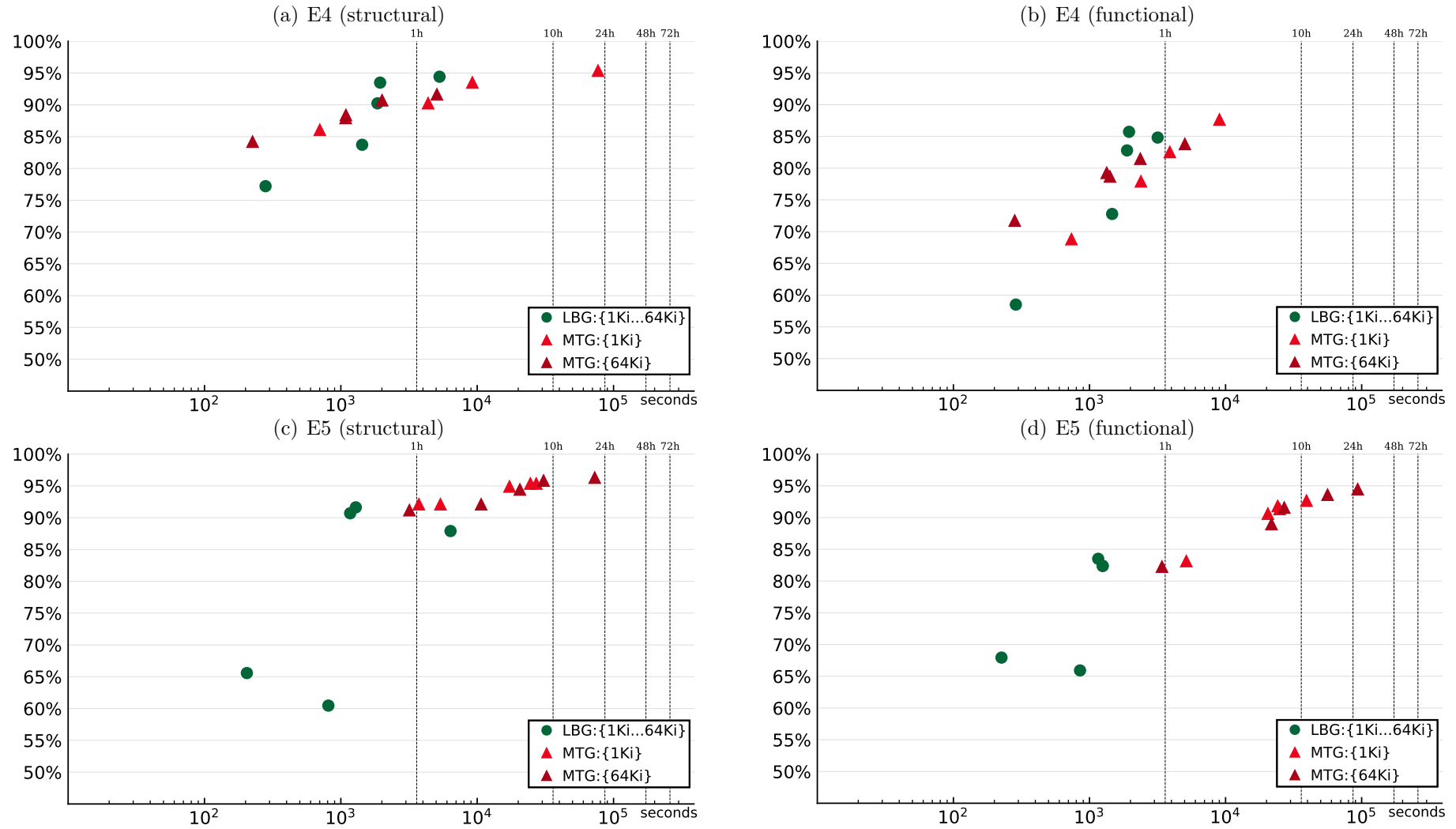


Figure 18 – Discovery constellations for 3-level MESI design errors



4.7 CONCLUSIONS AND PERSPECTIVES

The use of a Reinforcement Learning agent for solving some target problem requires two *problem-specific* key tailorings: (1) how to *represent the actions* and (2) how to *represent the state* of the environment. This chapter has addressed the first tailoring, and has indicated that the second one deserves further research.

This chapter proposed a canonical test-program representation for developing directed test generators targeting the verification of shared-memory behavior with MCM checkers. The theoretical guarantees mean that our canonical representation precludes the generation of redundant tests, and it prevents the limitation of test suites. The experimental evidence indicates that our representation provides proper grounds for defining actions, because it encodes effective primal behaviors. Since many actions could be defined by combining primal behaviors, larger sets of abstract actions can be envisaged. This paves the way towards the design of sophisticated agents for DTG. That is why we expect that future test generators for shared-memory verification can benefit from our test representation.

5 MULTICORE ENVIRONMENT STATE REPRESENTATION FOR AGENT-DIRECTED TEST GENERATION

This chapter reproduces the paper: *Bruno D. Miranda, Luiz M. V. Pereira, Márcio Castro, and Luiz C. V. dos Santos. Multicore Environment State Representation for Agent-Directed Test Generation. In: Proceedings of the 62nd ACM/IEEE Design Automation Conference (DAC 2025), ACM/IEEE, 2025.* It proposes a novel Reinforcement Learning approach to Directed Test Generation (DTG) for the verification of multicore systems. Its key contribution is the introduction of a state representation based on execution witnesses, enabling Reinforcement Learning agents to effectively recover environment state from raw observations. By encoding execution instances as unique signatures, the method provides a compact and non-ambiguous representation of multicore behaviors, which allows agents to achieve faster coverage evolution and error diagnosis compared to existing DTG techniques.

5.1 INTRODUCTION

The verification of the interactions between cores in multicore systems requires the execution of concurrent test programs until coverage goals are met, and no errors are left in the design. Such interactions through shared memory must obey a *Memory Consistency Model* (MCM) (Adve; Gharachorloo, 1996; Nagarajan et al., 2020), which allows many valid behaviors for the same concurrent program, leading to multiple correct outcomes for the same input stimuli. The lack of a one-to-one golden model for the expected behavior makes the generation of self-checking tests very complex (Adir; Shurek, 2002; Adir et al., 2004).

A pragmatic alternative is to split verification into complementary tasks: a generator produces a suite of test programs, while an external checker analyzes whether or not the observed behavior during the execution of each program complies with the MCM (Saha et al., 1995; Manovit; Hangal, 2006; Roy et al., 2006; Shacham et al., 2008; Graf; Andrade; Santos, 2023). This allows the generator to focus on coverage improvement, whereas the checker focuses on error diagnosis. This chapter follows such strategy for test generation.

Various approaches to *Directed Test Generation* (DTG) have been reported for the target verification domain, such as Bayesian networks (Fine; Ziv, 2003), Euler tour (Qin; Mishra, 2012; Lyu et al., 2019), genetic programming (Elver; Nagarajan, 2016), local search (Andrade et al., 2020), and reinforcement learning (Pfeifer et al., 2020).

Reinforcement learning (RL) involves the interaction between an *agent* and an *environment*. The agent receives a *reward* signal, and takes *actions* depending on the environment *state*. In many practical cases, the agent can only observe partial information about the state. The choice of proper *observations* depends on the target environment, and it impacts performance.

This chapter proposes a novel reinforcement learning approach to DTG which focuses on proper state representation for environments containing a *design under verification* (DUV) that is a multicore system. We show (1) what should be observed, (2) how observations should be represented, and (3) how they should be used to recover state information.

Firstly, we propose the notion of *witness* to execution instances of a concurrent program

as a proper key to observing the response of a multicore environment to an action taken by an agent. To define a witness, we repurposed the well-known *reads-from* relation (Alglave et al., 2010) for recovering state information. Despite its standard use for the synthesis of litmus tests (Alglave et al., 2010; Lustig et al., 2017), that relation has never been cast before into a state representation for enabling RL agents to *direct* test generation.

Secondly, we show that test generation can be conveniently constrained to enable a provenly *unique representation* for every observable execution witness induced by a test suite.

Finally, we show how environment state information can be recovered from the bounded *history of action-observation pairs*.

This chapter is organized as follows. Section 5.2 discusses our motivation in face of related work. Section 5.3 describes the proposed approach to DTG. Section 5.4 formulates our proposal for suitable observation, proper representation, and state recovery. Section 5.5 reports our experimental evaluation, and Section 5.6 draws overall conclusions.

5.2 MOTIVATION IN FACE OF RELATED WORK

Since model-based DTG (Qin; Mishra, 2012; Lyu et al., 2019) is usually tied to protocol or architecture, data-driven approaches are preferred (Fine; Ziv, 2003; Elver; Nagarajan, 2016; Andrade et al., 2020; Pfeifer et al., 2020). One of them (Fine; Ziv, 2003) dynamically redefines directives for a random test generation engine by means of statistical inference. That approach is based on modeling the relationship between coverage information and test directives as Bayesian networks. Mcjammer (Wagner; Bertacco, 2008) employs multiple agents in distinct core domains that generate concurrent memory access patterns. Mcjammer is not a test program generator, but a data-driven verification environment that directly generates stimuli based on the coverage goals of multiple agents.

The McVersi Test Generator (MTG) (Elver; Nagarajan, 2016) relies on Genetic Programming, and it led to superior coverage and error discovery as compared to random test generation and litmus tests. MTG represents the state of the art for evolutionary DTG.

The Reinforcement Learning Generator (RLG) (Pfeifer et al., 2020) is the only RL approach to DTG reported so far for the target domain, and it obtained results competitive with MTG. Thus, MTG and RLG are qualified baselines for the evaluation of novel DTG approaches.

Most DTG techniques are *coverage-directed* test generators (Fine; Ziv, 2003; Elver; Nagarajan, 2016; Andrade et al., 2020; Pfeifer et al., 2020), where coverage is the only observation that is tracked from the DUV to direct the generation of new tests. However, coverage is not an injection, since the same fraction of the coverage goals may correspond to different subsets with same cardinality. Although the cumulative coverage captures the evolution of the verification process, it does not accurately track the actual trajectory of responses to the induced stimuli.

Thus, to be improved, DTG needs extra observations from the DUV, but such information is usually restricted to external checkers, because most generators rely on them for being able to focus on coverage improvement. Such extra observations should not be tied to MCM, coherence protocol or microarchitecture to avoid dependence on a specific verification task.

Such DUV-agnostic observations are naturally exploited by litmus tests (Nagarajan et al., 2020). Since litmus test generation (Alglave et al., 2010; Lustig et al., 2017) cannot provide coverage guarantees (Elver; Nagarajan, 2016), and coverage-directed generation is restricted due to limited observation, proper observations should be extracted from the DUV to guide novel approaches to DTG. This motivates the use of a sequence of domain-specific observations as a trajectory that can be used to recover the RL notion of environment state (Sutton; Barto, 2018).

Successful applications of RL used to require handcrafted tuning (Tesauro, 1995) to each specific problem until the Deep Q-Network (DQN) approach (Mnih et al., 2013) has shown that the combination of Q-learning (Watkins; Dayan, 1992), deep neural network, and experience replay (Lin, 1992) could handle a collection of different problems (video games) of the same application domain (Atari suite) without having to resort to problem-specific tuning. High performance was obtained with the same hyper-parameters, network architecture, and observations (raw video frames). Since then, several extensions to DQN have been reported (Hessel et al., 2018). This feature of recent agents is attractive to the verification of multicore systems, where different tasks are induced by a variety of coherence protocols, core counts, microarchitectures, and coverage metrics.

To benefit from RL, the raw input for the agent should capture primal shared-memory behaviors, which are specified as *order relations* in formal frameworks for litmus test generation (Alglave et al., 2010; Lustig et al., 2017) and MCM checking (Manovit; Hangal, 2006; Graf; Andrade; Santos, 2023). Among them, the *reads-from* relation is particularly convenient, because it allows order to be inferred from observed values.

The use of values for representing order was proposed as part of a profiling technique (Ball; Larus, 1996), where a signature encoded the order of basic blocks in an execution path of a sequential program. That idea inspired a technique for trace compaction (Lee; Bertacco, 2017), where signatures were used to capture the order of memory operations induced by concurrent programs. Such compaction technique inspired us to rely on signatures for environment *state recovery*, enabling the use of RL agents for the target verification domain.

5.3 THE PROPOSED APPROACH

5.3.1 Background in Brief

The interaction between agent and environment occurs at discrete time steps $0, 1, 2, \dots, t$. At a given time step t , the agent receives some representation of the environment state, S_t , and it responds with some action A_t . In the next time step, the environment sends to the agent a scalar reward signal R_{t+1} , and evolves to a new state S_{t+1} . Both R_{t+1} and S_{t+1} are determined as a consequence of action A_t . The agent learns from interaction how to take actions on the environment in order to maximize expected rewards. Thus, the interaction between agent and environment can be expressed as $\langle S_0, A_0, S_1, R_1, A_1, S_2, R_2, A_2, S_3, R_3, \dots, A_{t-1}, S_t, R_t \rangle$.

The interaction is usually formalized as a Markov decision process, where R_{t+1} and S_{t+1} can be defined only in terms of S_t and A_t . This means that a state must be able to completely capture all the interaction history that could be relevant for future decision making

(Sutton; Barto, 2018). Such ideal property often does not hold in practice, because the agent observes only partial information about the environment state. Thus, some technique is required to approximate a Markov state from observations.

To handle partial observability, we do not assume that the environment emits the actual state, but it exposes some observation of signals that depends on that state. We also assume that rewards can be obtained from observations. Under these assumptions, the interaction can be expressed as an alternating sequence of actions A_{t-1} and observations O_t , as follows: $\langle A_0, O_1, A_1, O_2, A_2, O_3, \dots, A_{t-1}, O_t \rangle$.

5.3.2 Agent-based Approach to DTG

Let P_t denote a concurrent *test* program, and let $P_1, P_2, \dots, P_t, \dots, P_T$ denote a *test suite*.

Typically, each concurrent program is executed a certain number of times to expose multiple *execution instances* that are likely to exhibit distinct shared-memory behavior. Let I denote a set of execution instances induced by a given program, which are obtained by running the program $|I|$ times, in such a way that each execution is differently perturbed due to the non-determinism of thread concurrency.

Coverage is quantified as a fraction of the coverage goals reached during the execution of the programs of a test suite. Let C_{t+1} be the cumulative coverage obtained by running I execution instances of every program of the sequence $P_1, P_2, \dots, P_t$.

DTG aims to synthesize a test suite $P_1, P_2, \dots, P_t, \dots, P_T$ that maximizes C_{T+1} . To cast DTG as an RL problem, we introduce the notion of *verification episode*, which comprises successive epochs, each defined by discrete time steps.

Each epoch between steps t and $t+1$ consists of three phases: test generation, multiple execution, and coverage evaluation, as follows. An action A_t induces the generation of a program P_t . Then P_t is executed $|I|$ times in the DUV. Finally, C_{t+1} is updated with the *individual* contribution of P_t to the cumulative coverage (i.e. the joint contribution of all its $|I|$ execution instances). This is used to define the reward obtained with P_t .

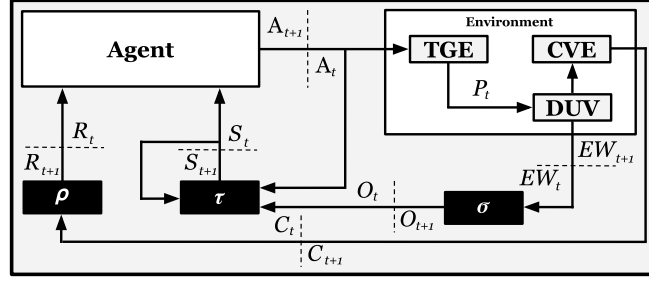
Definition 5.1. A *reward* is a function $\rho : \mathcal{R} \times \mathcal{R} \mapsto \mathcal{R}$ such that $R_{t+1} = \rho(C_{t+1}, C_t)$.

Fig. 19 illustrates our RL approach to DTG. The environment includes the DUV, a *test generation engine* (TGE), and a *coverage engine* (CVE), which computes the fraction of the coverage goals achieved by all instances of every program executed so far. Between environment and agent, three modules compute functions that convert actual environment information into the conventional interface expected by an RL agent.

Dotted lines represent temporal barriers to indicate the evolution between successive time steps. At a given time step t , when the environment is in state S_t , the action A_t induces the generation of a test program P_t . Then multiple instances of that program are executed in the DUV. At time step $t+1$, monitors in the DUV provide a *witness* EW_{t+1} to those executions, and the coverage engine provides the cumulative coverage C_{t+1} .

The function ρ converts coverage information into a reward R_{t+1} , and the function

Figure 19 – A reinforcement learning approach to DTG



σ converts witnessed information into an observation O_{t+1} . Both R_{t+1} and O_{t+1} are jointly obtained from the multiple executions of the test launched at step t .

The function τ computes a transition from the state S_t to the following state S_{t+1} for a given action-observation pair A_t, O_t .

The notion of execution witness and the functions σ and τ are the keys to our state representation, as follows.

5.4 THE PROPOSED STATE REPRESENTATION

Let us define *what* should be observed (to properly capture the environment's activity) and *how* it should be encoded (to ensure a non-ambiguous, compact representation).

5.4.1 Choice of suitable observation and proper representation

Let $\mathcal{OP}$ be the set of memory operations of a program. Let S_j^i denote that store operation $S_j \in \mathcal{OP}$ executes in core i . Let L_m^k denote that load operation $L_m \in \mathcal{OP}$ executes in core k .

Each test may induce different execution instances of a concurrent program, which can be seen as distinct orderings of the memory operations of a concurrent program (Gharachorloo, 1995).

Definition 5.2. An *execution instance*, written $<$, is a total order on the set $\mathcal{OP}$.

An execution instance $<$ determines which value is observed by each load. Let Val^0 denote the initial value assigned to some location. Let $Val[S]$ denote the value written by some store S , and let $Val_{<}[L]$ be the value read by some load L .

An execution instance $<$ induces the following relation.

Definition 5.3. L and S are in the *reads-from* relation, written $L \text{ rf } S$, iff $Val_{<}[L] = Val[S^i] \neq Val^0$.

The outcome of a non-synchronized concurrent program may serve as witness to a given execution instance. First, let us define a witness from the perspective of some core i . Such local witness relies on the outcome induced by the set $\mathcal{L}^i$ of load operations belonging to the thread executed by that core.

Definition 5.4. A core *witness* to execution instance $<$ is $EW_{<}^i = \{v : v = Val_{<}[L] \wedge L \in \mathcal{L}^i\}$.

Note that a witness is determined by which store (if any) is observed by each load: if there exists some S such that $L \text{ rf } S$, then $Val_{<}[L] = Val[S]$, otherwise $Val_{<}[L] = Val^0$.

Now let us define a core witness induced by a set I of execution instances of the same test program.

Definition 5.5. A core *witness* to execution instances I is:

$$EW_I^i = \bigcup_{< \in I} EW_{<}^i.$$

Let us define an aggregation of the witnesses at each core.

Definition 5.6. A multicore *witness* to execution instances I is $EW_I = (EW_I^1, EW_I^2, \dots, EW_I^i, \dots, EW_I^p)$.

To properly encode execution witnesses, let us constrain store value assignment: different natural numbers must be assigned to distinct stores, and 0 is reserved for initial values, as follows.

Constraint 1. $Val[S_j] \in \mathcal{N}$ for all $S_j \in \mathcal{OP}$.

Constraint 2. $Val^0 = 0$ for all locations.

Constraint 3. $Val[S_j] \neq Val[S_{m \neq j}] > 0$.

Such constraints are commonly applied to test generation, because unique values serve as non-ambiguous identifiers for stores, thereby reducing the complexity of MCM checkers (Manovit; Hangal, 2006).

A straightforward way to satisfy Constraint 3 *would* be to assign the next natural number to the next synthesized store, during the generation of an *individual test* program. However, to rule out ambiguity when observing execution witnesses induced by *different* programs, let us define a unified assignment policy for all programs of a *test suite*. The key is to assign unique values to stores, independently of the number of stores chosen for each thread. Let θ denote an upper bound for the number of stores in a thread, and let S_j^i be the j -th store of a given thread i . The following implies Constraint 3.

Constraint 4. $Val[S_j^i] = \theta(i-1) + j$ and $1 \leq j \leq \theta$ for all tests.

To encode witnesses under those constraints, let $\mathcal{EW}^i = \{EW_{<}^i\}$ denote the witnesses induced by every possible execution instance $<$, and let $\mathcal{P}(\mathcal{EW}^i)$ be its power set.

Definition 5.7. The signature of a witness to execution instances I at core i is a function $sig : \mathcal{P}(\mathcal{EW}^i) \mapsto \mathcal{N}$ such that:

$$sig(EW_I^i) = \sum_{v \in EW_I^i} v$$

Let us prove that a signature encodes a core witness properly.

Lemma 5.1. Under Constraints 1, 2, and 4, sig is an injection.

Proof: Let us consider the witnesses $EW_I^i, EW_{I'}^i \in \mathcal{P}(\mathcal{EW}^i)$, and let us assume that they are different. $EW_I^i \neq EW_{I'}^i$ implies $I \neq I'$. Thus, I and I' must have each at least one execution instance, say $<$ and $\prec$, which is not a member of the other, i.e.: $\{<\} \subseteq I \setminus I'$ and $\{\prec\} \subseteq I' \setminus I$. Therefore, $EW_I^i \neq EW_{I'}^i$ implies $EW_{<}^i \neq EW_{\prec}^i$ for some $<$ and $\prec$. If $EW_{<}^i \neq EW_{\prec}^i$ holds, there must exist at least one load $L \in \mathcal{L}^i$ such that $Val_{<}[L] = Val[S]$ and $Val_{\prec}[L] = Val[S']$ or $Val_{<}[L] = Val^o$. Since Constraint 4 implies Constraint 3, under Constraints 1, 2, and 3, we have $Val_{<}[L] \neq Val_{\prec}[L]$. For convenience, let $U = \{u : u \in EW_{I \cap I'}^i\}$, $V = \{v : v \in EW_{I \setminus I'}^i\}$, and $V' = \{v' : v' \in EW_{I' \setminus I}^i\}$. From Definition 5.7, we have: $sig(EW_I^i) = \sum_{u \in U} u + \sum_{v \in V} v$ and $sig(EW_{I'}^i) = \sum_{u \in U} u + \sum_{v' \in V'} v'$. Since $Val_{<}[L] \neq Val_{\prec}[L]$ holds for some L , it implies $\sum_{v \in V} v \neq \sum_{v' \in V'} v'$, which leads to distinct signatures. Since $EW_I^i \neq EW_{I'}^i$ implies $sig(EW_I^i) \neq sig(EW_{I'}^i)$, sig is an injection. ■

Let $\mathcal{EW} = \{EW_{<}\}$ be the set of *multicore* witnesses induced by every possible execution instance $<$.

Definition 5.8. The signature of a multicore witness to execution instances I is a function $\sigma : \mathcal{P}(\mathcal{EW}) \mapsto \mathcal{N}^p$ such that: $\sigma(EW_I) = (sig(EW_I^1), sig(EW_I^2), \dots, sig(EW_I^p))$.

Theorem 5.1. Under Constraints 1, 2, and 4, σ is an injection.

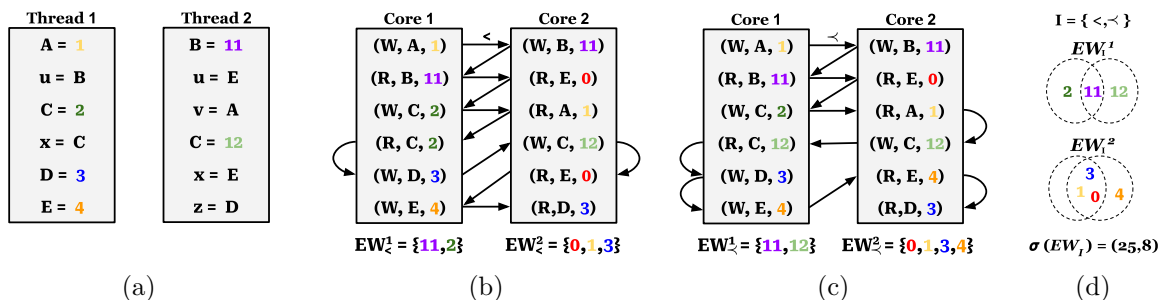
Proof: Let us consider the witnesses $EW_I, EW_{I'} \in \mathcal{P}(\mathcal{EW})$, and assume that they are different. If $EW_I \neq EW_{I'}$ holds, there must exist at least one core i such $EW_I^i \neq EW_{I'}^i$, which from Lemma 5.1 implies $sig(EW_I^i) \neq sig(EW_{I'}^i)$. From Definition 5.8, this is sufficient to conclude that $\sigma(EW_I) \neq \sigma(EW_{I'})$. Since $EW_I \neq EW_{I'}$ implies $\sigma(EW_I) \neq \sigma(EW_{I'})$, σ is an injection. ■

Theorem 5.1 guarantees that signatures are *unique* representations for observations of a multicore DUV.

5.4.2 Illustrative example

Fig. 20 illustrates the proposed representation for two execution witnesses of the same test program. Fig. 20(a) shows a test program with two threads, where capital letters denote shared memory locations, and small letters represent variables allocated in registers. Note that all values are natural numbers (Constraint 1), the locations are all initialized with zero

Figure 20 – An observation captured by a joint witness to execution instances $I = \{<, \prec\}$, and its representation as a signature.



(Constraint 2), and values written by stores are unique, greater than zero (Constraint 3), and obtained with $\theta = 10$ (Constraint 4).

Fig. 20(b) and Fig. 20(c) show distinct execution instances of that test program induced by different orders ($<$ and $\prec$). In each core, a trace captures the sequence of memory accesses resulting from the execution of load and store operations.

Each memory access is represented as a triple specifying the access type as write (W) or read (R), the location, and the value. Fig. 20(d) shows the joint witness and its signature. From now on, when we need to timestamp a witness, instead of $EW_{I,t}$, we simply write EW_t (assuming an implicit set I).

From now on, when we need to timestamp a witness, instead of $EW_{I,t}$, we simply write EW_t (assuming an implicit set I).

5.4.3 Recovering of state information from observations

Given the sequence $\langle A_0, O_1, A_1, O_2, A_2, O_3, \dots, A_{t-1}, O_t \rangle$, the recurrent pattern A_{t-1}, O_t is the basis for the approximation for the notion of state, where the interaction history is bounded.

Definition 5.9. For a bound $h \geq 1$, a state at time step $t \geq h$ is $S_t = \langle A_{t-h}, O_{t-h+1}, \dots, A_{t-2}, O_{t-1}, A_{t-1}, O_t \rangle$.

A proper value for the bound h can be obtained empirically.

Let $\mathcal{S}$, $\mathcal{A}$, and $\mathcal{O}$ denote the sets of states, actions, and observations. A transition from state S_t to S_{t+1} can be computed by shifting the representation of S_t one pair to the left, and appending A_t, O_{t+1} , so as to keep history bounded (Sutton; Barto, 2018).

Definition 5.10. A *transition* is a function $\tau: \mathcal{S} \times \mathcal{A} \times \mathcal{O} \mapsto \mathcal{S}$ such that $S_{t+1} = \langle A_{t-h+1}, O_{t-h+2}, \dots, A_{t-1}, O_t, A_t, O_{t+1} \rangle$.

We propose a state representation where $O_t = \sigma(EW_t)$.

5.5 EXPERIMENTAL EVALUATION

We built agent-based generators with the same set of actions. The *Random Agent Generator* (RAG) selects every action with the same probability. The *Deep-Q Generator* (DQG) relies on the DQN agent (Hessel et al., 2018), and it has variants with distinct state representations, as shown at the top of Table 10.

RAG rules out learning effects (because its agent’s policy is independent of state), serving as a baseline for evaluating the impact of state representation on learning, whether using general-purpose observations (coverage values in DQG⁺) or the proposed domain-specific observations (execution witnesses in DQG^{*}).

We compared RAG and DQG variants with directed test generators described in related works: the Reinforcement Learning Generator (RLG) (Pfeifer et al., 2020), whose state representation is not based on action-observation history, and the McVersi Test Generator (MTG)

Table 10 – Generators used to evaluate our state representation

Generator	Agent	State Representation	
		Observation	Trajectory record
RAG	random	n.a.	n.a.
DQG ⁺	DQN (Mnih et al., 2013)	C_t	action-observation history
DQG*		$\sigma(EW_t)$	action-observation history
RLG	(Pfeifer et al., 2020)	C_t	t
MTG	none	n.a.	n.a.

(Elver; Nagarajan, 2016), which does not rely on an agent, but on evolutionary directed test generation, as shown at the bottom of Table 10.

We evaluated each generator when performing different *verification tasks* targeting ARMv8 DUVs. Each task is obtained as a combination of distinct core counts (16 and 32) with different coverage metrics (structural transition coverage, as defined in (Elver; Nagarajan, 2016), and functional transition coverage, as defined in (Andrade et al., 2020)).

We relied on gem5’s infrastructure (Binkert et al., 2011) for simulation and design representation (we chose *O3*, *Ruby*, and *simple* for CPU, memory, and network models, respectively). For the DUVs, we adopted 64KiB (4-way) private L1 caches and a 4MiB (16-way) shared L2 cache, all with same block size (64 bytes), under a 2-level MOESI directory-based coherence protocol.

We ran experiments under the same range constraints on the number of operations ($n_{min} = 1Ki$, $n_{max} = 4Ki$) and shared locations ($s_{min} = 4$, $s_{max} = 128$). The allowed values within those ranges had to comply with generator’s differences¹.

RAG, DQG, and RLG can generate programs with *variable* numbers of operations in the same test suite, but MTG does not: all programs must have a *fixed* number of operations. That is why we report individual MTG results for range extremes.

DQG employs a DQN agent, which uses a *Q-network* for non-linear approximation of the state-action value function. We adopted a feed-forward Q-network containing two hidden layers with 256 neurons each (ReLU and tanh were used for activation in successive layers). At each discrete time step t , the agent maximizes the expected discounted return $G_t = \sum_{t'=t+1}^T \gamma^{t'-(t+1)} R_{t'}$. The values chosen for the hyper-parameters were: discount rate $\gamma = 0.9$, learning rate $\eta = 0.025$, batch size $\beta = 32$, and the probability of taking a random action was linearly scaled down from $\epsilon = 0.9$ (when the episode starts) down to $\epsilon = 0.05$ (when it reaches one hour). We adopted $h = 4$ as history bound and the reward function defined by Equation 5.1.

$$R_{t+1} = \rho(C_{t+1}, C_t) = \begin{cases} +1 & \text{if } C_{t+1} > C_t \\ -1 & \text{otherwise} \end{cases} \quad (5.1)$$

The number of execution instances of each test was set to five, and was induced as suggested in Elver & Nagarajan (2016). In each verification episode, every test was run five

¹ MTG requires the static definition of a *single* address-space constraint, which is called *test memory size* (TM). To accommodate such difference, we report MTG results under the (best) static constraint defined in (Elver; Nagarajan, 2016), i.e. TM=8KiB. We preserved all genetic parameters as they were originally set.

times under different simulation states (not related to the test itself) in such a way that the distinct executions of the same test were all perturbed differently.

Runtimes were measured on a Linux Server (Intel Xeon E5-2640 v4, 2.4 GHz, with 128GB of main memory).

5.5.1 Coverage Evolution with Time

We performed independent 24-hour verification episodes for each task. We launched each generator five times with distinct random seeds, and reported median coverage values. For DQG and RLG, a new set of random weights was used to initialize their neural networks at each new launching.

Fig. 21 shows the coverage evolution for different tasks. RLG led to the slowest coverage evolution. The state representations used by RLG and DQG^+ both rely on coverage values as generic observations. However, RLG does not take the history of action-observation pairs into account to approximate the environment state as DQG^+ does. This explains why RLG exhibits limited coverage evolution as compared to DQG^+ .

Note that DQG^* was superior to RAG for all tasks, while DQG^+ was mostly superior to RAG, except for Task 1. This indicates that the history of action-observation pairs *did* enable learning, which is more prominent when our state representation is used.

DQG^* was superior to DQG^+ , despite environment changes from one task to another. Thus, DQG^* observations capture relevant information about the environment *independently of task*, i.e. they are properly tailored to the target *domain*.

As compared to related works, DQG^* had the fastest coverage evolution for all tasks with exactly the same agent. Under the proposed state representation, the agent was able to learn on-line how to handle different tasks from “raw” observations of the environment, i.e. observations that were *not* represented in terms of specialized features handcrafted for any given task. Thus, a learning agent can make a directed test generator *effective* and *reusable* for distinct tasks of the same domain.

5.5.2 Impact on Error Discovery

We evaluated the impact of our state representation on the runtime required to expose errors in faulty designs. For the experiments, we selected challenging errors described in the literature, namely F2 and F3 (Graf; Andrade; Santos, 2023). We modified our DUVs to obtain faulty designs, each containing a single error. Since coverage results have shown that DQG^* dominates DQG^+ and MTG dominates RLG, we focused on comparing the best DQG version (DQG^*) with the best-performing related work (MTG).

Figure 21 – Median (cumulative) coverage evolution for 24-hour verification episodes

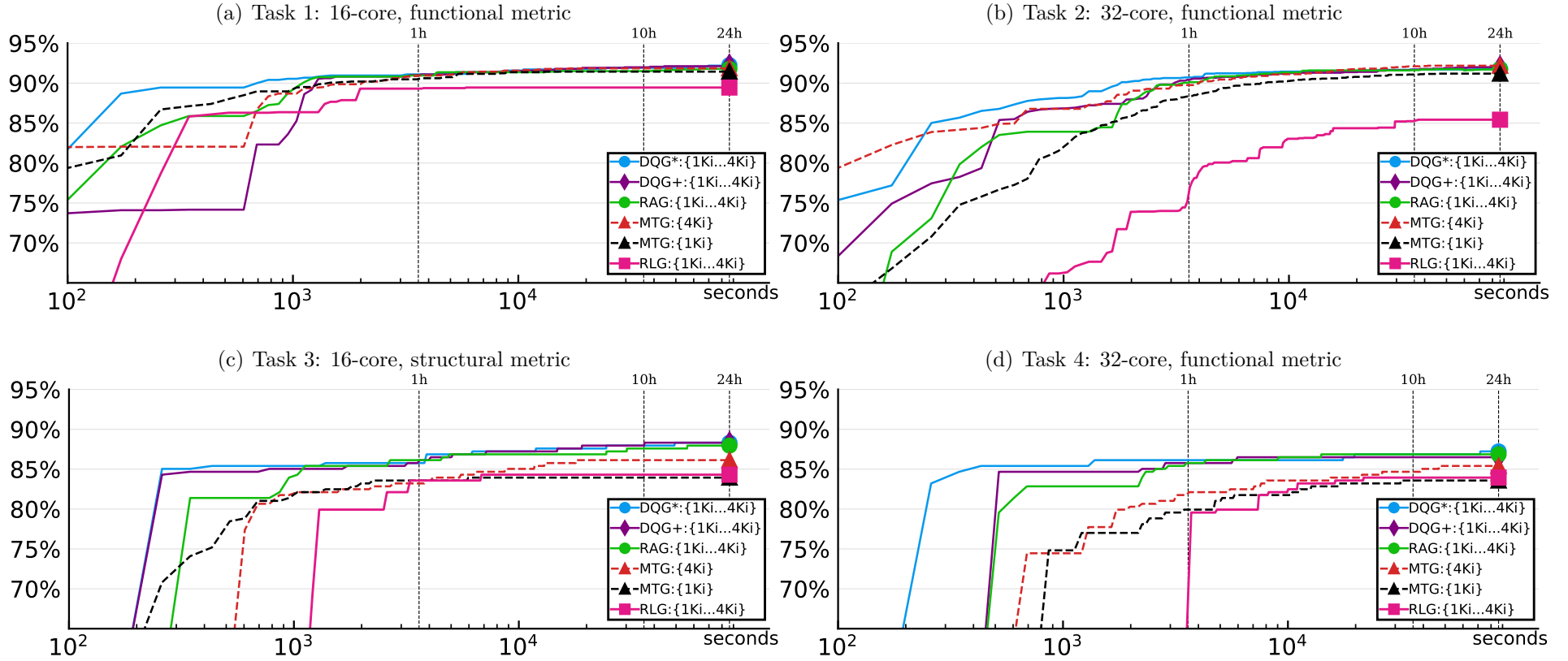


Figure 22 – Coverage-time error discovery constellations for faulty designs

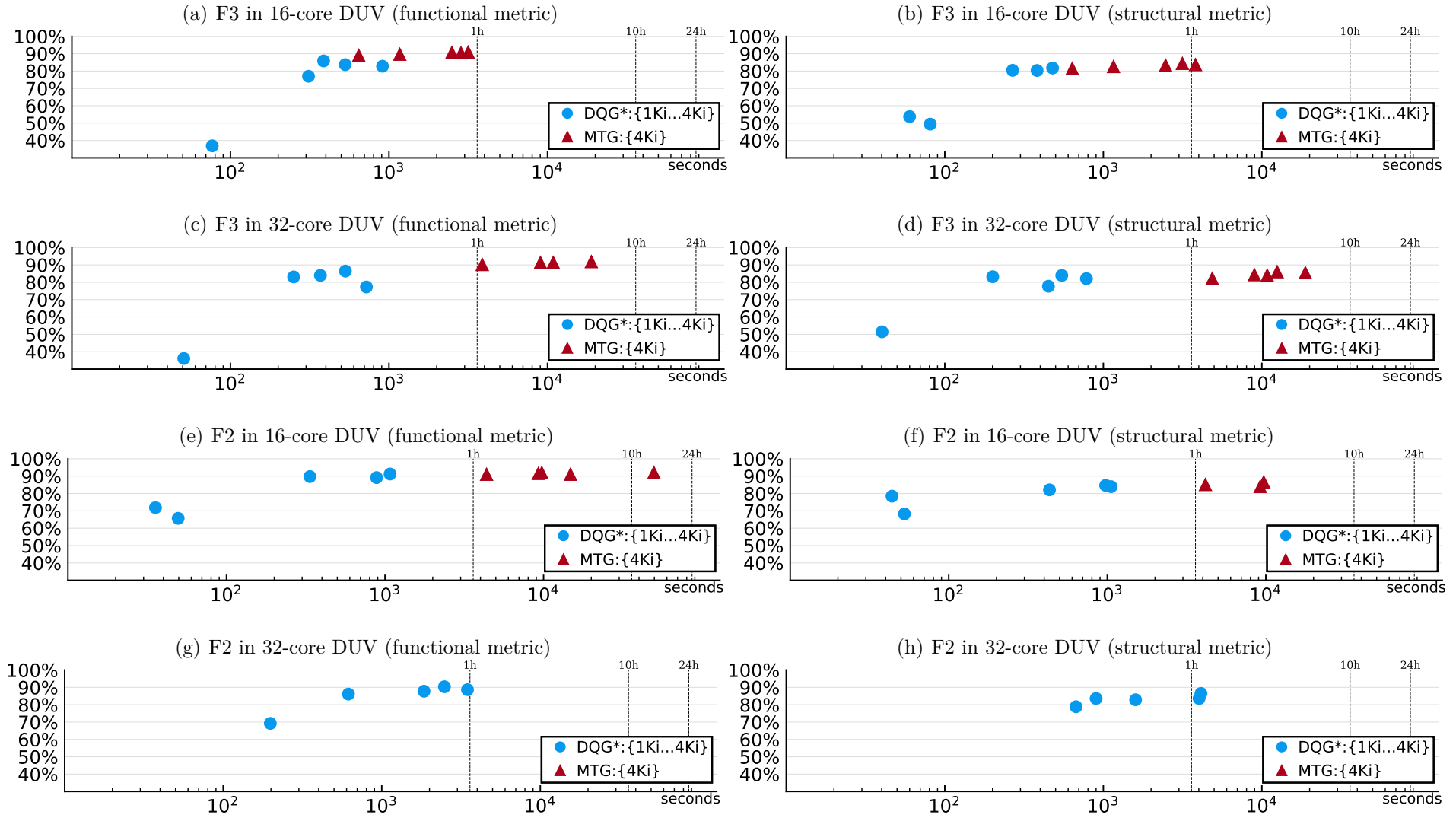


Table 11 – Median error discovery times (in seconds)

Error	Cores	Functional metric		Structural metric	
		DQG*	MTG	DQG*	MTG
F3	16	386	2493	267	2470
	32	369	9948	450	10802
F2	16	335	9744	431	9216
	32	1848	-	1597	-

We launched five verification episodes with distinct random seeds. We measured the runtime until the error was discovered and the coverage at the time of discovery. Then we plotted a constellation of coverage-time points for each generator. Fig. 22 shows constellation plots for faulty designs with a single error under a given coverage metric.

DQG* exposed every error in all 24-hour episodes, but MTG not always did. For a 32-core DUV, MTG did *not* expose F2 in *any* episode. When MTG exposed an error, it took much longer, as shown in Table 11.

DQG* led to the fastest error diagnosis in all scenarios, it exposed errors with lower coverage (on average), and its discovery times were less sensitive to the increase in core count. This indicates that the proposed state representation leads to high-quality tests, quite independently of coverage metric.

Higher-quality tests can be obtained when an agent directs generation instead of an evolutionary algorithm, mainly because the former is farsighted, while the latter is myopic. The former takes decisions based on expected returns over the long run. The latter takes them based on immediate returns (i.e. the best for the current population, regardless of next generations).

5.6 CONCLUSIONS

This chapter proposed a novel reinforcement learning approach to directed test generation for a crucial step of the design of a multicore system: the verification of the interaction between cores through shared memory. To be effective, that approach should rely on a proper representation for the state of a multicore environment, which requires the tracking of the history of action-observation pairs, and proper tailoring of observations to the target domain. This allows the agent-based generator to learn online how to be effective and reusable for *different* tasks of the *same* domain.

Results have shown that, when encoded as unique signatures, observations capturing the *reads-from* relation led to faster coverage evolution and faster error diagnosis, as compared to related work. This makes our approach to directed test generation a promising alternative to conventional techniques that are not based on agents.

6 CONCLUSIONS AND PERSPECTIVES

Shared memory is the primary communication mechanism in multicore processors, supported by a highly complex hardware structure. Designing such a structure is inherently challenging, error-prone, and demands sophisticated verification techniques. Since test programs execute significantly more slowly on design simulations, the use of DTG is essential to enable effective pre-silicon validation. Within the scope of functional verification of shared-memory behavior, this thesis focuses on DTG as a means to accelerate coverage evolution, thereby reducing the effort required to expose faulty designs.

Existing DTG techniques reported in the literature employ diverse test representations. Some are overly generic and cannot directly encode shared-memory behavior (Corno; Cumani; Squillero, 2003; Adir et al., 2004); others are tied to specific DUVs (Wagner; Bertacco, 2008; Qin; Mishra, 2012; Lyu et al., 2019); while some rely on RTG directives (Andrade; Graf; Santos, 2016; Andrade; Graf; dos Santos, 2019). Ideally, a test representation targeting shared-memory verification should be independent of any DUV, detached from specific DTG techniques, capable of mapping different addresses to the same cache line, and flexible enough to capture variable test sizes. To this end, this thesis introduced a canonical test-program representation for developing DTGs aimed at verifying shared-memory behavior with MCM checkers. Both theoretical guarantees and experimental evidence demonstrated that this representation enables thorough exploration of the search space. Its ability to encode fundamental shared-memory behaviors also facilitates the formulation of effective actions, allowing a learning agent to exploit coverage estimates more efficiently.

Most existing DTG techniques are *coverage-driven*, where fractions of coverage goals serve as the only observable feedback from the design (Fine; Ziv, 2003; Elver; Nagarajan, 2016; Andrade et al., 2020; Pfeifer et al., 2020). While cumulative coverage is useful for monitoring verification progress, it does not capture the trajectory of responses to stimuli generated by the test suite, since coverage is not an injective mapping. Ideally, a state representation for shared-memory verification should be independent of any DUV, capture fundamental shared-memory behavior, and leverage generation constraints to produce a unique representation of every observable execution induced by a test suite. This thesis proposed a novel environment state representation that records the history of agent–environment interactions using task-agnostic observations. Executions are encoded as unique signatures derived from the *reads-from* relation. Experimental results across four distinct verification tasks showed that this approach enables agents to learn online, achieving both effectiveness and reusability across different designs.

A significant portion of this thesis builds upon research conducted in collaboration with co-authors, parts of which have been published in the following venues (Miranda et al., 2025; Miranda; Castro; Santos, 2025):

- Bruno D. Miranda, Luiz M. V. Pereira, Márcio Castro, and Luiz C. V. dos Santos. **Multi-core Environment State Representation for Agent-Directed Test Generation**. In: *Proceedings of the 62nd ACM/IEEE Design Automation Conference (DAC 2025)*, ACM/IEEE, 2025.
- Bruno D. Miranda, Márcio Castro, and Luiz C. V. dos Santos. **A Canonical Test**

Representation for Verification of Shared-Memory Behavior in Multiprocessor Systems. *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, ACM, 2025. DOI: 10.1145/3762184.

6.1 FUTURE WORK

The contributions presented in this thesis open new directions and opportunities for further research and technical improvement:

- **Impact of representations on distinct DTG techniques:** Modern Electronic Design Automation (EDA) tools increasingly incorporate ML methods, underscoring their potential application in DTGs. The contributions proposed in Chapters 4 and 5, although independent of any specific DUV or DTG technique, were deliberately applied in the context of RL, as discussed in Section 2.2. Future work could explore the impact of these contributions when combined with other ML paradigms.
- **Impact of hyper-parameter tuning:** An in-house methodology¹ inspired by Nielsen (2015) was used to select the hyper-parameters in Chapter 5. Although effective for the target problem, this empirical approach has inherent limitations. For example, it deliberately omits statistical confidence: each hyper-parameter was selected based on a single one-hour verification episode with one random seed and one coverage metric. Additionally, it ignores parameters that may strongly influence learning, such as the agent–environment history bound (set to four) and the number of execution instances (set to five). Since hyper-parameters for RL, state representation, and neural networks are interdependent in DTG learning, future work could automate this process by leveraging multiple coverage metrics and modern open-source tools (Bergstra; Yamins; Cox, 2013; Moritz et al., 2017; Akiba et al., 2019).
- **Impact of the *reads-from* relation across multiple architectures:** Although the proposed state representation in Chapter 5 improves environment observability, the execution witnesses rely exclusively on the *reads-from* relation. Since signatures are unique, future work could investigate the inclusion of additional order relations (e.g., *from-reads*, *coherence order*). Moreover, all experimental results in Chapter 5 were obtained on tasks targeting the ARMv8 architecture and the MOESI protocol. Future research could extend the approach to other multicore architectures and coherence protocols (e.g., MOSI, MESI).
- **Impact of temporal order:** The predefined canonical collections of colliding, competing, and type subsets are unordered. In contrast, the thread subset is inherently ordered according to the program order. To expose non-deterministic behaviors, we impose an arbitrary ordering on all threads belonging to the same test program and execute the program multiple times. The primary objective is to vary the relative ordering of instructions across executions, thereby increasing the likelihood of conflicts and improving cumulative

¹ Detailed in Appendix A.

coverage. An alternative approach to addressing this limitation would be the introduction of an action category capable of dynamically modifying the thread execution order.

BIBLIOGRAPHY

- ADIR, A. et al. Genesys-Pro: innovations in test program generation for functional processor verification. **IEEE Design Test of Computers**, v. 21, n. 2, p. 84–93, Mar 2004. ISSN 0740-7475.
- ADIR, A.; SHUREK, G. Generating concurrent test-programs with collisions for multi-processor verification. In: **7th IEEE Int. High-Level Design Validation and Test Workshop**. Cannes, France: IEEE, 2002. p. 77–82.
- ADVE, S. V.; GHARACHORLOO, K. Shared memory consistency models: a tutorial. **Computer**, IEEE, v. 29, n. 12, p. 66–76, 12 1996.
- AKIBA, T. et al. Optuna: A next-generation hyperparameter optimization framework. In: **The 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining**. Anchorage, AK, USA: Association for Computer Machinery, 2019. p. 2623–2631.
- ALGLAVE, J. et al. Fences in weak memory models. In: **Computer Aided Verification**. Edinburgh, UK: Springer, 2010. v. 6174, p. 258–272.
- ANDRADE, G. A. G.; GRAF, M.; dos Santos, L. C. V. Chaining and Biasing: Test Generation Techniques for Shared-Memory Verification. **IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems**, IEEE, 2019.
- ANDRADE, G. A. G. et al. Steep Coverage-Ascent Directed Test Generation for Shared-memory Verification of Multicore Chips. In: **IEEE/ACM Int. Conf. on Computer-Aided Design (ICCAD)**. San Diego, CA, USA: IEEE, 2018. p. 29:1–29:8. ISBN 978-1-4503-5950-4.
- ANDRADE, G. A. G. et al. A directed test generator for shared-memory verification of multicore chip designs. **IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems**, v. 39, n. 12, p. 5295–5303, 2020.
- ANDRADE, G. A. G.; GRAF, M.; SANTOS, L. C. V. dos. Chain-Based Pseudorandom Tests for Pre-Silicon Verification of CMP Memory Systems. In: **34th IEEE International Conference on Computer Design (ICCD)**. Scottsdale, AZ, USA: IEEE, 2016. p. 552–559.
- BALL, T.; LARUS, J. Efficient path profiling. In: **Proceedings of the 29th Annual IEEE/ACM International Symposium on Microarchitecture. MICRO 29**. Paris, France: IEEE, 1996. p. 46–57.
- BELLEMARE, M. G.; DABNEY, W.; MUNOS, R. A distributional perspective on reinforcement learning. In: PRECUP, D.; TEH, Y. W. (Ed.). **Proceedings of the 34th International Conference on Machine Learning**. Sydney, Australia: PMLR, 2017. (Proceedings of Machine Learning Research, v. 70), p. 449–458.
- BELLMAN, R. **Dynamic Programming**. Princeton, NJ, USA: Princeton University Press, 1957.
- BERGSTRA, J.; YAMINS, D.; COX, D. D. Making a science of model search: Hyperparameter optimization in hundreds of dimensions for vision architectures. In: DASGUPTA, S.; MCALLESTER, D. (Ed.). **Proceedings of the 30th International Conference on Machine Learning (ICML 2013)**. Atlanta, GA, USA: JMLR Workshop and Conference Proceedings, 2013. v. 28, n. 3, p. 115–123.
- BINKERT, N. et al. The gem5 simulator. **SIGARCH Comput. Archit. News**, ACM, New York, NY, USA, v. 39, n. 2, p. 1–7, Aug 2011. ISSN 0163-5964.

CORMEN, T. H. et al. **Introduction to Algorithms, Third Edition**. 3rd. ed. Cambridge, MA, USA: The MIT Press, 2009. ISBN 0262033844.

CORNO, F.; CUMANI, F.; SQUILLERO, G. Exploiting auto-adaptive μ gp for highly effective test programs generation. In: **Evolvable Systems: From Biology to Hardware**. Berlin, Germany: LNCS, vol. 2606, Springer, 2003. p. 262–273. ISBN 978-3-540-36553-2.

ELVER, M. **McVerSi Framework**. online: GitHub, 2016. <https://github.com/melver/mc2lib>.

ELVER, M.; NAGARAJAN, V. McVerSi: A test generation framework for fast memory consistency verification in simulation. In: **IEEE Int. Symp. on High Perform. Computer Architecture (HPCA)**. Barcelona, Spain: IEEE, 2016. p. 618–630.

FINE, S. et al. Harnessing machine learning to improve the success rate of stimuli generation. **IEEE Transactions on Computers**, IEEE, v. 55, n. 11, p. 1344–1355, 2005.

FINE, S.; ZIV, A. Coverage Directed Test Generation for Functional Verification Using Bayesian Networks. In: **ACM Design Automation Conference (DAC)**. Anaheim, CA, USA: IEEE, 2003. p. 286–291. ISBN 1-58113-688-9.

FORTUNATO, M. et al. **Noisy Networks for Exploration**. 2017. Disponível em: <https://arxiv.org/abs/1706.10295>.

FREITAS, L. S.; RAMBO, E. A.; SANTOS, L. C. V. dos. On-the-fly verification of memory consistency with concurrent relaxed scoreboards. In: **Design, Automation, and Test in Europe (DATE)**. Grenoble, France: IEEE, 2013. p. 631–636. ISBN 978-1-4503-2153-2.

GHARACHORLOO, K. **Memory consistency models for shared-memory multiprocessors**. Tese (Doutorado) — Stanford University, Stanford, CA, USA, 1995.

GHARACHORLOO, K. et al. Memory consistency and event ordering in scalable shared-memory multiprocessors. **SIGARCH Comput. Archit. News**, 1990.

GRAF, M.; ANDRADE, G. A. G.; SANTOS, L. C. V. dos. Evecheck: An event-driven, scalable algorithm for coherent shared memory verification. **IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems**, v. 42, n. 2, p. 683–696, 2023.

GREENGARD, S. Will risc-v revolutionize computing? **Commun. ACM**, Association for Computing Machinery, New York, NY, USA, v. 63, n. 5, p. 30–32, abr. 2020. ISSN 0001-0782. Disponível em: <https://doi.org/10.1145/3386377>.

HANGAL, S. et al. TSOtool: A program for verifying memory systems using the memory consistency model. **ACM SIGARCH Comp. Arch. News**, v. 32, n. 2, p. 114–123, 2004. ISSN 0163-5964.

HASSELT, H. v.; GUEZ, A.; SILVER, D. Deep reinforcement learning with double q-learning. In: **Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence**. Phoenix, AZ, USA: AAAI Press, 2016. (AAAI'16), p. 2094–2100.

HENNESSY, J. L.; PATTERSON, D. A. **Computer Architecture: A Quantitative Approach**. 6th. ed. Cambridge, MA, USA: Morgan Kaufmann Publishers Inc., 2017. ISBN 0128119055.

HESSEL, M. et al. Rainbow: Combining improvements in deep reinforcement learning. In: **Proceedings of the AAAI Conference on Artificial Intelligence**. New Orleans, LA, USA: AAAI Press, 2018. v. 32.

HU, W. et al. Linear Time Memory Consistency Verification. **IEEE Transactions on Computers**, v. 61, n. 4, p. 502–516, Apr 2012. ISSN 0018-9340.

IOANNIDES, C.; EDER, K. I. Coverage-directed test generation automated by machine learning – a review. **ACM Trans. Des. Autom. Electron. Syst.**, Association for Computing Machinery, New York, NY, USA, v. 17, n. 1, jan 2012. ISSN 1084-4309.

LAASCH, M. To us, risc-v is the architecture of the next one to two decades. **ATZelectronics worldwide**, v. 20, n. 9, p. 14–19, 2025. Disponível em: <https://doi.org/10.1007/s38314-025-2051-9>.

LAMPORT, L. Time, clocks, and the ordering of events in a distributed system. **Communications of the ACM**, 1978.

LEE, D.; BERTACCO, V. Mtracecheck: Validating non-deterministic behavior of memory consistency models in post-silicon validation. In: **2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA)**. Toronto, ON, Canada: Association for Computing Machinery, 2017. p. 201–213.

LIN, L.-J. **Reinforcement learning for robots using neural networks**. Tese (Doutorado) — Computer Science, Pittsburgh, PA, USA, 1992. UMI Order No. GAX93-22750.

LUSTIG, D. et al. Automated Synthesis of Comprehensive Memory Model Litmus Test Suites. In: **ACM Int. Conf. on Arch. Support for Progr. Lang. and Op. Systems (ASPLOS)**. Xi'an, China: Association for Computing Machinery, 2017. p. 661–675. ISBN 978-1-4503-4465-4.

LYU, Y. et al. Directed test generation for validation of cache coherence protocols. **IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems**, v. 38, n. 1, p. 163–176, Jan 2019. ISSN 0278-0070.

MANOVIT, C.; HANGAL, S. Completely verifying memory consistency of test program executions. In: **IEEE Int. Symposium on High-Performance Computer Architecture (HPCA)**. Austin, TX, USA: IEEE, 2006. p. 166–175.

MIRANDA, B. D.; CASTRO, M.; SANTOS, L. C. d. A canonical test representation for verification of shared-memory behavior in multiprocessor systems. **ACM Trans. Des. Autom. Electron. Syst.**, Association for Computing Machinery, New York, NY, USA, v. 30, n. 5, set. 2025. ISSN 1084-4309.

MIRANDA, B. D. et al. Multicore environment state representation for agent-directed test generation. In: **2025 62nd ACM/IEEE Design Automation Conference (DAC)**. San Francisco, CA, USA: IEEE, 2025. p. 1–7.

MNIH, V. et al. Asynchronous methods for deep reinforcement learning. In: BALCAN, M. F.; WEINBERGER, K. Q. (Ed.). **Proceedings of The 33rd International Conference on Machine Learning**. New York, NY, USA: PMLR, 2016. (Proceedings of Machine Learning Research, v. 48), p. 1928–1937.

MNIH, V. et al. Playing atari with deep reinforcement learning. **CoRR**, abs/1312.5602, 2013. Disponível em: <http://arxiv.org/abs/1312.5602>.

MORITZ, P. et al. Ray: A distributed framework for emerging AI applications. **CoRR**, abs/1712.05889, 2017. Disponível em: <http://arxiv.org/abs/1712.05889>.

NAGARAJAN, V. et al. **A Primer on Memory Consistency and Cache Coherence, 2nd Edition**. San Rafael, CA, USA: Morgan & Claypool Publishers, 2020. (Synthesis Lectures on Computer Architecture).

NIELSEN, M. A. **Neural Networks and Deep Learning**. San Francisco, CA, USA: Determination Press, 2015. v. 25.

PATTERSON, D. 50 years of computer architecture: From the mainframe cpu to the domain-specific tpu and the open risc-v instruction set. In: **2018 IEEE International Solid-State Circuits Conference - (ISSCC)**. San Francisco, CA, USA: IEEE, 2018. p. 27–31.

PFEIFER, N. **A reinforcement learning approach to directed test generation for shared memory verification**. Dissertação (Mestrado) — Federal University of Santa Catarina, Florianópolis, SC, Brazil, 2021.

PFEIFER, N. et al. A Reinforcement Learning Approach to Directed Test Generation for Shared Memory Verification. In: **Design, Automation & Test in Europe Conference & Exhibition (DATE)**. Grenoble, France: IEEE, 2020. p. 538–543. ISSN 1558-1101.

QIN, X.; MISHRA, P. Automated generation of directed tests for transition coverage in cache coherence protocols. In: **Design, Automation, and Test in Europe (DATE)**. Dresden, Germany: IEEE, 2012. p. 3–8. ISSN 1530-1591.

RAMBO, E.; HENSCHER, O.; SANTOS, L. dos. Automatic generation of memory consistency tests for chip multiprocessing. In: **IEEE Int. Conf. on Electronics, Circuits and Systems (ICECS)**. Beirut, Lebanon: IEEE, 2011. p. 542–545.

ROBBINS, H. Some aspects of the sequential design of experiments. **Bulletin of the American Mathematical Society**, v. 58, n. 5, p. 527–535, 1952. Disponível em: <https://doi.org/10.1090/S0002-9904-1952-09620-8>.

ROY, A. et al. **Fast and Generalized Polynomial Time Memory Consistency Verification**. Berlin, Germany: Springer Berlin Heidelberg, 2006. v. 4144. 503–516 p. (Lecture Notes in Computer Science, v. 4144). ISBN 978-3-540-37411-4.

RUMMERY, G. A.; NIRANJAN, M. **On-line Q-learning using connectionist systems**. Cambridge, UK, 1994. Disponível em: <http://www.inf.ed.ac.uk/teaching/courses/mlsc/lectures/rum96.pdf>.

SAHA, A. et al. A simulation-based approach to architectural verification of multiprocessor systems. In: **Proceedings International Phoenix Conference on Computers and Communications**. Scottsdale, AZ, USA: IEEE, 1995. p. 34–37.

SCHAUL, T. et al. **Prioritized Experience Replay**. 2016. Disponível em: <https://arxiv.org/abs/1511.05952>.

SHACHAM, O. et al. Verification of chip multiprocessor memory systems using a relaxed scoreboard. In: **IEEE/ACM International Symposium on Microarchitecture (MICRO 41)**. Como, Italy: IEEE, 2008. p. 294–305. ISBN 978-1-4244-2836-6.

SULLIVAN, M. et al. Multiprocessor design verification with generated realistic mp programs. In: **Proceedings International Phoenix Conference on Computers and Communications**. Scottsdale, AZ, USA: IEEE, 1995. p. 389–395.

SUTTON, R. S.; BARTO, A. G. **Reinforcement Learning: An Introduction**. Cambridge, MA, USA: A Bradford Book, 2018. ISBN 0262039249.

TESAURO, G. Temporal difference learning and td-gammon. **Commun. ACM**, Association for Computing Machinery, New York, NY, USA, v. 38, n. 3, p. 58–68, mar. 1995. ISSN 0001-0782. Disponível em: <https://doi.org/10.1145/203330.203343>.

WAGNER, I.; BERTACCO, V. MCjammer: Adaptive Verification for Multi-core Designs. In: **Design, Automation, and Test in Europe (DATE)**. Munich, Germany: Association for Computing Machinery, 2008. p. 670–675. ISSN 1530-1591.

WANG, Z. et al. Dueling network architectures for deep reinforcement learning. In: BALCAN, M. F.; WEINBERGER, K. Q. (Ed.). **Proceedings of The 33rd International Conference on Machine Learning**. New York, NY, USA: PMLR, 2016. (Proceedings of Machine Learning Research, v. 48), p. 1995–2003.

WATKINS, C. J. C. H. **Learning from delayed rewards**. Tese (Doutorado) — King's College, Cambridge, United Kingdom, 1989.

WATKINS, C. J. C. H.; DAYAN, P. Q-learning. **Machine Learning**, v. 8, n. 3–4, p. 279–292, 1992.

WOOD, D.; GIBSON, G.; KATZ, R. Verifying a multiprocessor cache controller using random test generation. **IEEE Design & Test of Computers**, v. 7, n. 4, p. 13–25, 1990.

APPENDIX A – METHODOLOGY TO SELECT THE HYPER-PARAMETERS OF THE NEURAL NETWORK

This methodology was inspired by the methodology proposed in Nielsen (2015), which was refined and adapted to the target problem. However, Nielsen’s methodology targets supervised learning. Since the neural network is used in the context of RL, the tuning of hyper-parameters has to take into account RL-specific issues.

A.1 METHODOLOGY GOAL

Obtain empirically suitable values for learning rate (η), batch size (β), the number of neurons (nn) for the ϵ -range and for a task set by running a single verification episode in each trial.

Hypothesis for large-scale experimentation: The values for the hyper-parameters which were obtained with a single episode of the *reduced problem* (small-sized range constraints) are likely to work for multiple episodes of the reduced problem and might work for the *real* problem (nominal range constraints).

A.2 ARBITRARY CHOICES

- Empirically made a compromise between immediate and future rewards, in order to offer a good convergence and practical performance ($\gamma = 0.9$);
- Same single random seed used for all trials.

A.3 HYPER-PARAMETER ϵ

The DQN agent uses a ϵ -greedy behavior policy to select actions while learning about the *target* policy. The behavior policy is applied by scheduling decreasing values of ϵ along multiple episodes: from an initial value ϵ_{start} to a final value ϵ_{end} . Thus, for the purposes of RL, suitable hyper-parameter values for the range $[\epsilon_{start}, \epsilon_{end}]$ must be selected. The methodology tunes the hyper-parameters *separately* for each extreme of that range, and then select suitable values for operating on the whole range. The DQG uses the same values for distinct tasks. Thus, the hyper-parameters must be tuned separately for each task, and then select suitable values for the whole task set.

A.4 STRIPING DOWN

A.4.1 The Q-Network

- Empirically determined input size and output layer equal to number of actions;
- Single (or multiple) hidden layer(s), each one with the same number of neurons.

A.4.2 The target problem

- Let the range of generation constraints be n_{min} , n_{max} , s_{min} and s_{max} ;
- Consider a set of verification tasks;
- Consider a time limit T for each episode;
- Let $|I|$ be the number of execution instances, fixed in both tasks.

A.5 SUITABLE HYPER-PARAMETER VALUES

To obtain suitable values, a preparation step (Step 0) and three tuning steps were performed. Each tuning step provokes a perturbation on a single hyper-parameter. If the perturbation leads to a signal of learning (i.e. a significant improvement w.r.t RAG), then the perturbed value is adopted, otherwise the original value is preserved. Multiple rounds of tuning steps might be repeated while significant improvements are observed. In Step 0, threshold values are obtained for each hyper-parameter to prevent the choice of values causing improper behavior.

A.5.1 Guidelines for validating a new value's impact on learning

Consider the following definitions:

- η_i , bs_i , nn_i are the values of η , bs , nn obtained in the end of Round i ;
- $t_{DQG}(\eta_i, bs_i, nn_i)$ is the time at which DQG reaches 90% of coverage when simultaneously selects the values η_i , bs_i , nn_i in the end of Round i .

The following shorthand notations were adopted:

- $t_{DQG}(i)$ denotes $t_{DQG}(\eta_i, bs_i, nn_i)$;
- $t_{DQG}(hp')$ denotes the time at which DQG reaches 90% when a modified value hp' was assigned to a single hyper-parameter at some Round i , while the others are kept unchanged.

A.5.1.1 Criterion for hyper-parameter update

The value of an hyper-parameter is updated only if leads to a new t_{DQG} that is at least 10% shorter than the old t_{DQG} .

A.5.2 Step Zero - threshold values

- Chosen arbitrary values for bs and nn not leading to improper behavior;
- Obtained an order-of-magnitude estimate for η_{th} ;
 - η_{th} is the value for the learning rate *above* which the behavior is improper¹;

¹ The value of an hyper-parameter leads to improper behavior if it causes increasing loss or "unacceptable" oscillations in loss function.

- Obtained a power-of-two estimate for bs_{th} ;
 - bs_{th} is the value for the batch size *below* which the behavior is improper;
- Obtained a power-of-two estimate for nn_{th} .
 - nn_{th} is the value of the number of neurons *below* which the behavior is improper.

A.5.3 Tuning routines

Algorithm 5 DQG-LEARNING-RATE-UPDATE(i, η_{i-1}, η_{th})

```

1: if  $i=1$  then
2:   return  $\eta_{th}/2$ 
3: else
4:    $\eta' \leftarrow \eta_{i-1}/2$ 
5:   if  $(t_{DQG}(i-1) - t_{DQG}(\eta')) \geq t_{DQG}(i-1)/10$  then
6:     return  $\eta'$  {it must not lead to improper behavior}
7:   else
8:     return  $\eta_{i-1}$ 
9:   end if
10: end if

```

Algorithm 6 DQG-BATCH-SIZE-UPDATE(i, bs_{i-1}, bs_{th})

```

1: if  $i=1$  then
2:   return  $bs_{th} * 2$ 
3: else
4:    $bs' \leftarrow bs_{i-1} * 2$ 
5:   if  $(t_{DQG}(i-1) - t_{DQG}(bs')) \geq t_{DQG}(i-1)/10$  then
6:     return  $bs'$  {it must not lead to improper behavior}
7:   else
8:     return  $bs_{i-1}$ 
9:   end if
10: end if

```

Algorithm 7 DQG-NEURONS-UPDATE(i, nn_{i-1}, nn_{th})

```

1: if  $i=1$  then
2:   return  $nn_{th} * 2$ 
3: else
4:    $nn' \leftarrow nn_{i-1} * 2$ 
5:   if  $(t_{DQG}(i-1) - t_{DQG}(nn')) \geq t_{DQG}(i-1)/10$  then
6:     return  $nn'$  {it must not lead to improper behavior}
7:   else
8:     return  $nn_{i-1}$ 
9:   end if
10: end if

```

Algorithm 8 SUITABLE-PARAMETERS-DQN($\eta_{th}, bs_{th}, nn_{th}$)

```

1:  $i \leftarrow 1$ 
2: repeat
3:    $\eta_i \leftarrow \text{DQG-LEARNING-RATE-UPDATE}(i, \eta_{i-1}, \eta_{th})$ 
4:    $bs_i \leftarrow \text{DQG-BATCH-SIZE-UPDATE}(i, bs_{i-1}, bs_{th})$ 
5:    $nn_i \leftarrow \text{DQG-NEURONS-UPDATE}(i, nn_{i-1}, nn_{th})$ 
6: until  $\eta_i = \eta_{i-1} \wedge bs_i = bs_{i-1} \wedge nn_i = nn_{i-1}$ 

```

A.6 HYPER-PARAMETER TUNING

A.6.1 For a given task

The following steps were adopted:

- Performed Step 0;
- Hyperparameter tuning (Algorithm 8):
 - applied for constant ϵ_{start} ;
 - applied for constant ϵ_{end} ;
- The following values were chosen for η , bs and nn :
 - $\eta = \min\{\eta(\epsilon_{start}), \eta(\epsilon_{end})\}$
 - $bs = \max\{bs(\epsilon_{start}), bs(\epsilon_{end})\}$
 - $nn = \max\{nn(\epsilon_{start}), nn(\epsilon_{end})\}$

A.6.2 For the task set

Tuned η as the learning rate jointly suitable for the task set {Task-1, Task-2}:

$$\eta = \min\{\eta(Task - 1), \eta(Task - 2)\}$$

Tuned bs as the batch size jointly suitable for the task set {Task-1, Task-2}:

$$bs = \max\{bs(Task - 1), bs(Task - 2)\}$$

Tuned nn as the neurons jointly suitable for the task set {Task-1, Task-2}:

$$nn = \max\{nn(Task - 1), nn(Task - 2)\}$$

The selected hyper-parameter values for DQG were reported in the Chapter 5.