Deteção Automática de Anomalias em Arquiteturas de Microsserviços

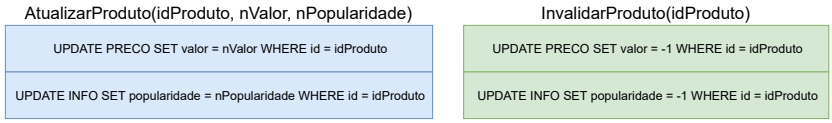
Valentim Romão, Rafael Soares, Vasco Manquinho e Luís Rodrigues

INESC-ID, Instituto Superior Técnico, U. Lisboa
{valentim.romao,
joao.rafael.pinto.soares,vasco.manquinho,ler}@tecnico.ulisboa.pt

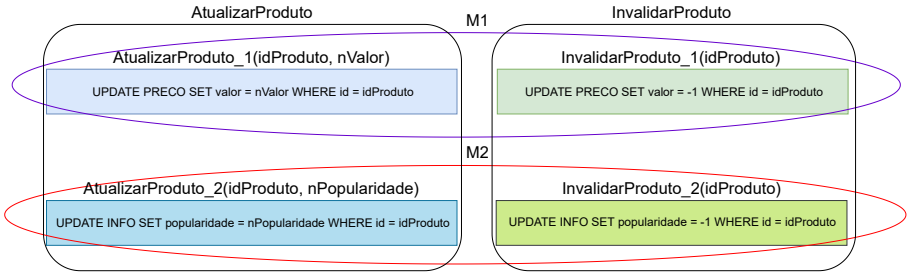
Resumo A arquitetura de microsserviços permite estruturar uma aplicação como um conjunto de serviços fracamente acoplados. Esta arquitetura tem várias vantagens, tais como modularidade e a capacidade de escala, que motivam a migração de monólitos para microsserviços, apesar dos desafios colocados pela falta de isolamento entre funcionalidades que requerem a invocação de vários microsserviços. Numa aplicação monolítica, cada funcionalidade é tipicamente executada como uma única transação que acede a uma única base de dados com propriedades ACID. Numa arquitetura de microsserviços, uma funcionalidade pode estar fracionada em múltiplas sub-transações independentes e cada uma pode ser executada por um microsserviço diferente. O intercalamento destas sub-transações, quando as funcionalidades se executam concorrentemente, pode levar a resultados inesperados, também chamados de anomalias. Neste trabalho apresentamos uma ferramenta capaz de detetar automaticamente estas anomalias. Apresentamos também uma avaliação experimental da nossa ferramenta, recorrendo a micro-testes e a duas decomposições em microsserviços da aplicação-padrão TPC-C.

1 Introdução

A arquitetura de microsserviços suporta o desenvolvimento de uma aplicação como um conjunto de serviços fracamente acoplados, contrastando com a tradicional arquitetura monolítica constituída por um único componente centralizado. Esta arquitetura tem várias vantagens quando comparada com a arquitetura monolítica. Em primeiro lugar, cada microsserviço apenas concretiza a lógica relacionada com um pequeno subconjunto de entidades geridas pela aplicação, tornando o código de cada microsserviço mais coeso e fácil de desenvolver e manter. Em segundo lugar, cada microsserviço pode ser desenvolvido de forma independente, permitindo um desenvolvimento mais ágil da aplicação como um todo. Em terceiro lugar, a arquitetura oferece uma maior flexibilidade na gestão do sistema, uma vez que os microsserviços podem ser geridos de forma independente. Devido a estas vantagens, várias empresas estão atualmente a usar a arquitetura de microsserviços quando desenvolvem as suas aplicações, e algumas estão inclusivamente a migrar as suas aplicações monolíticas para a arquitetura de microsserviços [1].



(a) Versão monólito.

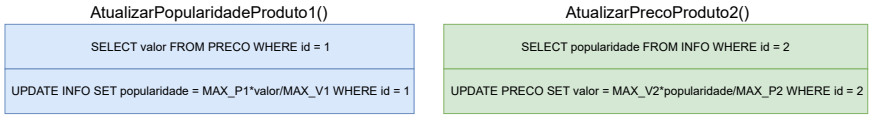


(b) Versão microsserviços.

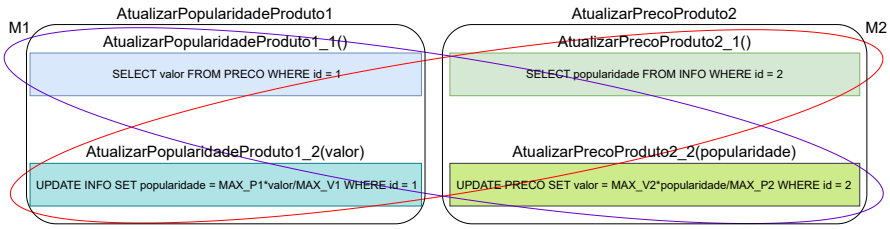
Figura 1: Caso de teste A.

No entanto, a arquitetura de microsserviços introduz novos desafios. Uma aplicação monolítica é tradicionalmente constituída por múltiplas funcionalidades, sendo cada uma descrita como uma transação executada contra uma única base de dados. Estas transações satisfazem as propriedades ACID (**A**tomicidade, **C**onsistência, **I**solamento, **D**urabilidade), assegurando o isolamento entre execuções concorrentes destas funcionalidades. Ao migrar uma aplicação monolítica para microsserviços, uma funcionalidade pode ser fracionada em várias sub-transações, sendo cada sub-transação possivelmente executada em microsserviços diferentes. Considere os casos nas Figuras 1 e 2, onde existem duas entidades, *Preco* e *Info*. Cada caso tem duas funcionalidades distintas, *AtualizarProduto* e *InvalidarProduto*, e *AtualizarPopularidadeProduto1* e *AtualizarPrecoProduto2*, respetivamente. Note-se que ao fazer um fracionamento com base nas entidades (Figuras 1b e 2b), cada funcionalidade origina sub-transações que executam em microsserviços diferentes. A entidade *Preco* é tratada no microsserviço M1 (elípse roxa) e a entidade *Info* no microsserviço M2 (elípse vermelha). A funcionalidade a que cada sub-transação pertence está representada pela caixa envolvente.

Tipicamente, os microsserviços utilizam o padrão de desenho *base de dados por serviço* [2]. Como tal, embora cada sub-transação seja isolada das restantes sub-transações que executam no mesmo microsserviço, a execução concorrente de funcionalidades pode levar a intercalamentos de sub-transações que executam em diferentes microsserviços, algo que não ocorria no monólito. Isto pode levar a resultados inesperados, também chamados de anomalias, que derivam de execuções não-serializáveis das transações. Para além disso, a execução de uma funcionalidade pode já não ser atómica, dado que cada sub-transação confirma individualmente, deixando-se assim de assegurar a atomicidade dos efeitos das operações de uma funcionalidade. Isto acontece nas Figuras 1 e 2, dado que cada



(a) Versão monólito.



(b) Versão microsserviços.

Figura 2: Caso de teste B.

funcionalidade passa a ser composta por duas sub-transações, e é possível a primeira sub-transação confirmar sem saber o desfecho da segunda sub-transação.

O número de anomalias que podem surgir durante a execução de funcionalidades em microsserviços depende de como o monólito é decomposto, em particular o número de microsserviços que interagem entre si e que entidades são geridas por cada microsserviço. Encontrar estas anomalias de forma automática pode ser bastante útil, tanto mais que as anomalias de concorrência são notoriamente difíceis de identificar através de testagem [3]. Isto é possível usando ferramentas que conseguem gerar todos os intercalamentos possíveis e que comparam as execuções que podem ocorrer no monólito com as execuções que podem ocorrer numa dada decomposição. Esta informação pode ajudar o programador a estimar qual será a decomposição menos problemática e fornecer pistas para o desenvolvimento do código que mitiga as anomalias detetadas. No entanto, esta tarefa não é trivial, pois têm de ser considerados não só os efeitos causados pelo fracionamento de uma transação numa sequência de sub-transações independentes, mas também os efeitos das sub-transações lerem versões mutuamente incoerentes de objetos remotos através do armazenamento local do seu microsserviço.

Neste artigo, apresentamos a ferramenta MAD (de *Microservices Anomaly Detector*) capaz de automaticamente detetar anomalias de serializabilidade (leitura suja, escrita suja, atualização perdida, atraso na escrita e atraso na leitura) que resultam da decomposição de uma aplicação monolítica em microsserviços, através da codificação do problema numa formula de Satisfabilidade Modulo Teorias (SMT) [1] e recorrendo ao Z3 [4] para encontrar atribuições satisfazíveis.

[1] Uma generalização de Satisfabilidade Proposicional e que usa fragmentos de Lógica de Primeira Ordem.

Para verificar o correto funcionamento, precisão e aplicabilidade da ferramenta, realizámos um conjunto de experiências, inicialmente utilizando um conjunto de casos de teste simples, para validarmos a capacidade de deteção de anomalias da ferramenta em decomposições de microsserviços simples, e de seguida aplicámos a ferramenta a um caso de teste mais complexo baseado no TPC-C.

2 Trabalho Relacionado

Agrupamos o trabalho relacionado em três classes distintas, a saber: i) ferramentas que não detetam anomalias mas fornecem uma estimativa do número de anomalias que podem ocorrer (tipicamente recorrendo a heurísticas); ii) ferramentas que detetam anomalias sem ter acesso ao código do programa, e; iii) ferramentas que detetam anomalias tendo acesso ao código fonte. De seguida, fazemos um sumário de cada umas destas abordagens.

2.1 Heurísticas para Estimar o Número de Anomalias

Existem várias propostas de algoritmos para estimar o número de anomalias que podem ocorrer ao executar uma dada decomposição de um monólito em microsserviços. Estes algoritmos são baseados em heurísticas e pretendem ajudar o programador a escolher a melhor decomposição para a migração, considerando que o esforço de programação associado a uma decomposição será proporcional ao número de anomalias estimadas. Dentro desta categoria, destacamos os trabalhos apresentados em [5] e [6]. Estes trabalhos usam heurísticas que recorrem à análise do código fonte para identificar padrões de acesso a entidades que podem expor estados intermédios e gerar anomalias, por exemplo os casos em que uma funcionalidade executa diferentes microsserviços. Uma vantagem destas heurísticas é que podem ser concretizadas de forma muito eficiente e aplicadas a monólitos com um número muito elevado de linhas de código. Uma desvantagem é fornecerem apenas valores aproximados, que podem ser por omissão ou por excesso consoante a heurística. Em particular, podem apresentar falsos positivos dado não possuírem uma granularidade suficientemente fina para distinguir que alguns acessos são feitos a instâncias diferentes de uma mesma entidade.

2.2 Deteção de Anomalias usando Abordagens Caixa Preta

As abordagens do tipo "caixa preta" para a deteção de anomalias não precisam de ter acesso ao código do programa que estão a analisar. O procedimento destas ferramentas passa por enviar valores de entrada e analisar os valores de saída, e com base nessa informação determinar se ocorreram anomalias, através das diferenças entre o comportamento observado e o comportamento esperado. Ferramentas como o Cobra [7] e MonkeyDB [8] são exemplos desta abordagem. Estas ferramentas conseguem ser mais genéricas que as heurísticas apresentadas anteriormente, uma vez que não necessitam de ter acesso ao código do sistema

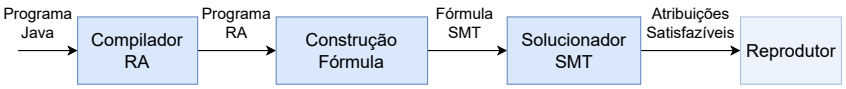


Figura 3: Esquema do processamento do CLOTHO.

analisado. Por outro lado, a análise pode ser incompleta, caso na geração de valores de entrada e no agendamento das operações executadas nunca seja gerada uma combinação que desencadeie um comportamento anómalo do sistema.

2.3 Deteção de Anomalias usando Abordagens Caixa Branca

Por último, analisamos trabalhos que utilizam uma abordagem do tipo "caixa branca" para detetar anomalias, ou seja, têm em conta o código fonte do programa na sua análise. Estas ferramentas primeiro extraem informação do programa, tal como as transações, os tipos dos parâmetros, a ordem de execução, entre outros, e, com base nessa informação, consideram todas as execuções do programa que possuem anomalias. Exemplos de ferramentas que usam esta abordagem são o ANODE [9], o CLOTHO [10] e o CLOTHO+ [11]. Estas ferramentas têm duas vantagens em relação às ferramentas anteriores, nomeadamente, não geram falsos positivos e conseguem fazer uma análise completa, que deteta todas as anomalias que podem ocorrer. As principais desvantagens destas ferramentas residem na necessidade de ter acesso ao código fonte do sistema a ser testado e na sua complexidade temporal e espacial, devido ao tempo e espaço de memória necessários para gerar todas as possíveis execuções anómalas do sistema.

Dado que o CLOTHO serve de base para o nosso trabalho, apresentamos com mais pormenor o funcionamento desta ferramenta. O CLOTHO [10] é uma ferramenta desenhada para analisar transações de um sistema de armazenamento replicado (em diferentes centros de dados), de modo a detetar anomalias de serializabilidade que podem ocorrer quando as réplicas são atualizadas segundo semânticas de coerência fracas. O CLOTHO tem adicionalmente a capacidade de reproduzir as anomalias em execuções concretas. O funcionamento desta ferramenta baseia-se em criar uma fórmula SMT, na qual as atribuições satisfazíveis correspondem a grafos cíclicos em que os vértices são operações e as arestas são as relações entre as operações, sendo que estes grafos representam execuções não serializáveis das transações [12]. O processo está dividido em três fases de análise e uma última fase de reprodução das anomalias, como ilustrado na Figura 3.

A ferramenta recebe um programa em linguagem Java e converte-o para uma Representação Abstrata (RA) semelhante a SQL. Este passo simplifica a extração de informação do programa, assim como a identificação de arestas entre operações. Os tipos de aresta possíveis são: ST (mesma transação); RW (leitura seguida de escrita); WR (escrita seguida de leitura); e WW (escrita seguida de escrita). Com base na RA do programa, a ferramenta constrói uma fórmula SMT. Esta fórmula possui cinco conjuntos de restrições, sendo estes os seguintes: $\varphi_{context}$ para representar os valores que são possíveis de estar na base de

dados; φ_{db} para representar as garantias de coerência e isolamento oferecidas pela base de dados; $\varphi_{dep\rightarrow}$ para representar as arestas de dependência entre as operações que pertencem ao ciclo de dependência da anomalia; $\varphi_{\rightarrow dep}$ para representar as arestas de dependência entre operações que não estão no ciclo de dependência da anomalia; $\varphi_{anomaly}$ limita as estruturas das possíveis anomalias a um número máximo de transações executadas sequencialmente, a um número máximo de transações executadas paralelamente e a um comprimento máximo para o ciclo de dependência. Um solucionador SMT (no caso do CLOTHO o Z3 [4]) é utilizado para encontrar as atribuições que satisfazem a fórmula. De notar que cada atribuição satisfazível da formula SMT corresponde a uma anomalia no programa. Caso contrário, se a formula SMT não for satisfazível, então o programa não tem anomalias dentro dos limites definidos para a construção da fórmula. Por último, o CLOTHO permite ainda simular a execução concreta das anomalias detetadas, no entanto este aspeto não é abordado no nosso trabalho.

O CLOTHO+ [11] inclui três extensões ao CLOTHO para modelar parcialmente o comportamento dos microsserviços, nomeadamente, a concretização dos modelos de coerência formalizados no CLOTHO, anotações para indicar em qual microsserviço cada transação executa e a introdução de uma relação para indicar que duas operações pertencem à mesma classe/tipo de transação.

2.4 Comparação

A Tabela 1 apresenta uma comparação entre as ferramentas anteriormente referidas. Dividimos a tabela em duas secções. A primeira secção (Propriedades) refere-se às caraterísticas das ferramentas. Dividimos esta secção em três colunas que capturam, respectivamente, a abordagem usada pela ferramenta para analisar um sistema (Análise), as garantias de coerência que a ferramenta está à espera que o sistema a ser testado respeite, considerando aquelas a que a ferramenta pode ser estendida a suportar (Modelo de Coerência) e se a ferramenta está orientada a analisar a arquitetura de microsserviços (Orientado a Microsserviços?). A segunda secção (Técnicas) refere-se aos mecanismos utilizados pelas ferramentas para cumprir o seu propósito. Esta secção dividimos também em três colunas que capturam, respectivamente, como a ferramenta identifica a existência de anomalias (Método), se utiliza um solucionador SMT (Solucionador SMT?) e qual o tipo de execuções que a ferramenta considera (Execuções Analisadas). De notar que a nossa ferramenta oferece uma análise Caixa Branca, aplicável a qualquer modelo de coerência, e que tem em conta todos os aspetos da arquitetura de microsserviços, algo que as outras ferramentas não fazem.

3 Ferramenta

O "Microservices Anomaly Detector" (MAD) é uma ferramenta que, como o próprio nome sugere, foi desenvolvida para detetar as anomalias que podem ocorrer ao executar uma decomposição de um monólito em microsserviços. Para desenvolver esta ferramenta, usámos como ponto de partida o CLOTHO [10]

Tabela 1: Comparação entre ferramentas.

	Propriedades			Técnicas		
	Análise	**Modelo de Coerência**	**Orientado a Microsserviços?**	**Método**	**Solucionador SMT?**	**Execuções Analisadas**
Complexity Metric	Heurística	Consistência Eventual	Sim	Análise Estática	Não	-
Metrics Refinement	Heurística	Consistência Eventual	Sim	Análise Estática	Não	Abstratas
Cobra	Caixa Preta	Serializabilidade	Não	Testagem	Sim	Abstratas
MonkeyDB	Caixa Preta	Qualquer	Não	Testagem	Não	Concretas
ANODE	Caixa Branca	Qualquer	Não	Análise Estática	Sim	Abstratas
CLOTHO	Caixa Branca	Qualquer	Não	Análise Estática	Sim	Abstratas/ Concretas
CLOTHO+	Caixa Branca	Qualquer	Parcial	Análise Estática	Sim	Abstratas
MAD	Caixa Branca	Qualquer	Sim	Análise Estática	Sim	Abstratas

com vários modelos de coerência [11]. Escolhemos o CLOTHO devido às suas vantagens em relação às outras ferramentas e ao facto do acesso ao código fonte poder ser útil para objetivos complementares ao nosso trabalho, tais como a escolha da melhor decomposição. A elevada complexidade temporal e espacial pode representar uma limitação do nosso trabalho, no entanto na Secção 4.2 demonstramos que a ferramenta consegue ser aplicada a aplicações como o TPC-C. A ferramenta CLOTHO não consegue fazer a análise que pretendemos, uma vez que não captura os aspetos relacionados com a migração para a arquitetura de microsserviços, tais como perceber que a funcionalidade do monólito, que executa numa única transação, ao ser executada numa decomposição em microsserviços, pode corresponder à execução em sequência de várias sub-transações. Para além disso, o CLOTHO não permite especificar que transações diferentes podem ser executadas em microsserviços diferentes (cada microsserviço pode ser modelado no CLOTHO como um centro de dados, mas seria necessário que o CLOTHO permitisse definir quais as transações que executam em cada centro de dados). De notar que a forma como a decomposição é gerada é ortogonal ao nosso trabalho, e que já existem ferramentas na literatura para ajudar nesse processo [13,14,15].

3.1 Noções de Transações Fracionadas e de Microsserviços

A primeira extensão feita à ferramenta consiste na adição das noções de microsserviços e de transações fracionadas (sub-transações). Para tal, adaptamos a anotação do trabalho CLOTHO+ [11] para se chamar "ChoppedTransaction" e possuir dois campos: "originalTransaction" para indicar qual era a transação que existia originalmente e que foi fracionada para surgir aquela sub-transação, e "microservice" para indicar qual o microsserviço em que a transação/sub-transação vai executar. Por exemplo, considerando a Figura 1b, cada sub-transação da funcionalidade *AtualizarProduto* teria uma anotação "ChoppedTransaction", na qual o campo "originalTransaction" teria o valor "AtualizarProduto" e o campo "microservice" teria os valores "M1" e "M2", respetivamente.

Através da informação do campo "originalTransaction" é possível saber que, embora as transações possam ser consideradas independentes, continua a exis-

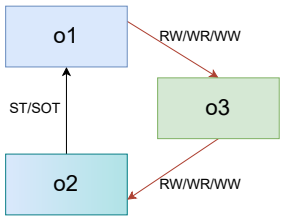


Figura 4: Definição de ciclo com anomalia no MAD.

tir uma ligação entre sub-transações que pertencem à mesma funcionalidade. Com base neste conhecimento, criamos um novo tipo de aresta SOT (**S**ame **O**riginal **T**ransaction) para indicar que duas transações pertencem à mesma instância de uma funcionalidade. Esta nova aresta é semelhante à aresta ST (**S**ame **T**ransaction) já existente no CLOTHO. Uma das semelhanças reside no facto de que com estas noções é possível estabelecer qual será a ordem de execução de cada operação da funcionalidade, tanto ao nível das operações que pertencem à mesma transação, assim como entre operações de sub-transações diferentes, sendo esta ordem de execução determinada através da ordem em que as operações estão definidas no código do programa a ser testado. Esta noção é necessária, uma vez que as sub-transações irão ser executadas pela sua ordem original, de modo a manter o comportamento original da funcionalidade, tal como podemos comprovar pelas Figuras 1 e 2. Outra semelhança é o impacto que esta aresta tem na definição de um ciclo que representa uma anomalia. No CLOTHO, um ciclo com anomalia tem de possuir pelo menos uma aresta ST e pelo menos duas arestas de dependência (RW, WR, WW). No MAD, estendemos esta condição para possuir a nova aresta que introduzimos (SOT) sendo atualmente um ciclo com anomalia caracterizado por ter pelo menos uma aresta ST ou SOT e pelo menos duas arestas de dependência (Figura 4). No entanto, as arestas ST e SOT diferem nas propriedades assumidas, sendo que nas arestas ST existe isolamento das operações, enquanto que nas arestas SOT não é garantido esse isolamento, de modo que é possível o intercalamento com outras operações.

A informação do campo "microservice" é usada na análise para influenciar a visibilidade entre operações, de modo a replicar a propagação dos valores entre microsserviços diferentes e quando é tornada visível a atualização de um valor.

3.2 Adaptações ao CLOTHO

Para além das alterações e adições anteriormente descritas, tivemos de fazer várias adaptações ao CLOTHO para este suportar as semânticas que pretendíamos, tais como o facto do armazenamento poder não ser distribuído, como assumimos que aconteça no caso do monólito, assim como a possibilidade de uma escrita não ser sempre visível para uma leitura posterior na mesma partição, dado que as operações podem ter sido executadas em microsserviços diferentes.

Para a primeira adaptação, como queremos também simular as condições de um monólito, alteramos a análise para, por omissão, ser feita considerando ape-

nas uma réplica. Para tal, acrescentamos condições que implicitamente assumem que se estará a correr as transações em apenas um nó de armazenamento e que consideram quais as linhas das tabelas que são acedidas para reproduzir o comportamento de trincos e outros mecanismos, no contexto de um monólito. Isto é feito através do uso de uma implicação que define que se três operações, o_1, o_2 e o_3 acedem ao mesmo objeto, o_1 e o_2 pertencem à mesma funcionalidade, o_2 acontece depois de o_1, e o_3 vê o valor de o_1, então o_3 terá de ver o valor de o_2.

A segunda adaptação resulta do facto de precisarmos de restringir uma condição de visibilidade indireta, cujo objetivo era representar que se uma operação de escrita acontecesse antes de uma operação de leitura, então a operação de escrita seria sempre visível para a leitura se ambas acontecessem em centros de dados da mesma partição. A nossa alteração restringiu esta regra para apenas ser aplicada caso as duas operações estivessem no mesmo microsserviço.

3.3 Modelos de Coerência

De modo a fazer uma análise fidedigna com o ambiente em que os sistemas irão executar, assumimos que entre operações que executam no mesmo microsserviço será utilizada serializabilidade e que entre operações de microsserviços diferentes será utilizada consistência eventual [16]. Para simular estes modelos, utilizamos as condições dos modelos de coerência definidas no artigo do CLOTHO e implementadas no CLOTHO+, com alguns ajustes para serializabilidade ser apenas aplicada entre operações que executam no mesmo microsserviço.

3.4 Apresentação de Anomalias por Tipo

Por último, acrescentamos também uma funcionalidade extra para mostrar o número de anomalias encontradas por cada um dos tipos canónicos de anomalias (leitura suja, escrita suja, atualização perdida, atraso na escrita e atraso na leitura). A técnica usada para o fazer consiste em criar um conjunto de padrões considerando apenas as arestas entre as operações e, quando uma anomalia é encontrada, verificar a qual dos padrões aquela anomalia corresponde. Caso a anomalia seja mais complexa e as suas arestas não correspondam a nenhum dos padrões, então a anomalia é considerada do tipo "Outro".

4 Avaliação

A nossa avaliação foca-se na comparação entre os resultados que podem ser obtidos através de heurísticas como a descrita no trabalho "A Complexity Metric for Microservices Architecture Migration" [5] (apresentado também na Tabela 1 e que passaremos a designar por CMMAM) e os resultados obtidos pela nossa ferramenta MAD. Para este efeito, recorremos a duas aplicações monolíticas que decompomos em microsserviços. A primeira aplicação foi concebida para fins ilustrativos e exemplificar as diferenças entre as duas ferramentas. A segunda aplicação é baseada numa aplicação definida no projeto OLTP-Bench [17].

4.1 Decomposição de uma Micro-Aplicação

Consideramos uma micro-aplicação desenhada à medida para facilitar a ilustração das diferenças entre o MAD e a CMMAM. Esta micro-aplicação possui duas tabelas, *Preco* e *Info*, sendo que a tabela *Preco* é pública e indica o preço de cada item, tendo as colunas *id* (chave primária) e *valor*, e a tabela *Info* é privada e possui dados apenas disponíveis para a gerência da loja, tendo as colunas *id* (chave primária) e *popularidade*. Neste contexto, criamos dois casos de teste ilustrados nas Figuras 1 e 2. O caso A é composto por duas funcionalidades: *AtualizarProduto* para atualizar o preço e a popularidade de um produto; e *InvalidarProduto* para invalidar tanto o preço como a popularidade de um produto (colocar ambos a -1). O caso B é composto também por duas funcionalidades: *AtualizarPopularidadeProduto1* para atualizar a popularidade do produto com identificador 1 consoante o seu preço; e *AtualizarPrecoProduto2* para atualizar o valor do preço do produto com identificador 2 consoante a sua popularidade.

Na primeira experiência, utilizamos o caso de teste A (Figura 1). Neste caso de teste existem problemas na decomposição de monólito para microsserviços, dado que passa a poder existir intercalamentos entre as sub-transações das duas funcionalidades, algo que não era possível no monólito. Dado que a métrica de complexidade apenas contabiliza operações inversas (para cada escrita considera o número de leituras e vice-versa), neste caso, como existem apenas escritas, a métrica irá devolver que a complexidade das funcionalidades é zero. Por outro lado, ao utilizar o MAD, uma vez que este considera as dependências entre escritas, foram detetadas as seguintes quatro anomalias: i) a invalidação de um produto (*InvalidarProduto*) ser feita entre as execuções de *AtualizarProduto_1* e *AtualizarProduto_2*, criando um cenário em que um produto pode ter o preço inválido mas a popularidade não estar inválida; ii) a atualização de um produto (*AtualizarProduto*) ser feita entre as execuções de *InvalidarProduto_1* e *InvalidarProduto_2*, criando um cenário em que um produto pode ter a popularidade inválida mas o preço não estar inválido; iii) a atualização de um produto (*AtualizarProduto*) ser feita concorrentemente com outra atualização do mesmo produto, podendo este intercalamento levar a valores de preço e popularidade incoerentes; e iv) a invalidação de um produto (*InvalidarProduto*) ser feita concorrentemente com outra invalidação do mesmo produto, sendo as escritas realizadas sobre estados que não eram visíveis na versão monólito das transações.

Na segunda experiência, utilizamos o caso de teste B (Figura 2). Neste caso de teste não existem problemas, visto que cada funcionalidade acede a objetos diferentes, embora das mesmas entidades. Isto não é um problema, dado que os acessos são feitos a instâncias diferentes, de modo que não se irá aceder a um estado intermédio do mesmo objeto que apenas passou a estar visível devido ao fracionamento das transações. No entanto, como a CMMAM apenas considera a entidade (classe de domínio) acedida para o seu cálculo, não será tido em conta que as funcionalidades vão aceder a instâncias diferentes, levando assim a métrica a indicar que a complexidade total das funcionalidades é quatro. Por oposição, o MAD considera as linhas a que cada operação acede, conseguindo assim fazer uma análise que tem em conta que cada funcionalidade acede a

uma linha diferente, o que resulta no número de anomalias detetadas a ser zero, demonstrando que aquela decomposição não irá introduzir problemas.

4.2 Decomposição do TPC-C

O TPC-C é uma aplicação definida no projeto OLTP-Bench [17] que simula o comportamento de um sistema de encomendas e gestão de armazém. Esta aplicação é frequentemente usada para avaliar sistemas transacionais, tendo sido também usada na avaliação do CLOTHO. Esta aplicação possui cinco funcionalidades: colocar uma encomenda; realizar um pagamento; verificar o estado da última encomenda de um cliente; processar encomendas ainda não entregues; e verificar o número de produtos que têm um nível de estoque abaixo de um dado limite. Para a avaliação do MAD, fizemos as adaptações necessárias para ser processada pela ferramenta e obtermos três versões da sua concretização, nomeadamente, uma versão monólito com as transações não fracionadas (*mono*) e duas versões que correspondem a decomposições distintas em microsserviços (*decomposição-1* e *decomposição-2*): na primeira decomposição cada entidade é gerida por um microsserviço diferente e na segunda cada microsserviço agrega mais que uma entidade, tendo em conta os acessos feitos por cada funcionalidade.

Na Tabela 2, apresentamos os resultados de aplicar as ferramentas CMMAM e MAD à aplicação TPC-C. Estes resultados demonstram que a ferramenta MAD consegue ser aplicada a sistemas com várias transações e tabelas. Para além disso, também é possível realçar o potencial do MAD para ajudar na escolha da decomposição em microsserviços menos problemática. Em relação aos resultados obtidos, as duas ferramentas apontam para a *decomposição-2* ser a melhor decomposição entre as duas, dado que a CMMAM apresenta uma complexidade de 36 para a *decomposição-1* e de 20 para a *decomposição-2*, e o MAD apresenta 68 anomalias para a *decomposição-1* e 25 anomalias para a *decomposição-2*. No entanto, o rácio entre os problemas de cada decomposição varia consoante a ferramenta, visto que o MAD considera também as dependências entre escritas, conseguindo obter um valor mais preciso dos problemas de cada decomposição.

Na nossa análise, optamos por utilizar o valor 4 para o comprimento máximo do ciclo, representando que nos grafos que o MAD procura podem haver no máximo 4 arestas. O processo de escolha deste valor passou por começar com o valor 3, visto que, segundo a definição, é o número mínimo de arestas necessário para ter um ciclo com anomalia, e fomos incrementando até conseguirmos obter um número significativo de anomalias num tempo de resposta razoável, convergindo assim para o valor 4. Na Tabela 3, apresentamos as anomalias detetadas pelo MAD divididas por tipo, o que permite ao programador antecipar possíveis situações anómalas. Outro aspeto a salientar é o número relativamente alto de anomalias que estão na coluna "#Outros", em particular na *decomposição-1*. Isto acontece, pois os padrões que consideramos possuem o número mínimo de operações para descrever cada tipo de anomalia. Por exemplo, numa leitura suja, basta existirem duas escritas numa funcionalidade e uma leitura noutra para ser possível uma execução em que se leia um valor intermédio. No entanto, caso

Tabela 2: Resultados do TPC-C.

	#Transações	#Tabelas	#Microsserviços	Ferramenta	Complexidade/ #Anomalias	Tempo de Execução [s]
TPC-C mono	5	9	1	CMMAM	0	$\simeq 0$
				MAD	0	33
TPC-C decomposição-1	22	9	9	CMMAM	36	$\simeq 0$
				MAD	68	13.241
TPC-C decomposição-2	12	9	3	CMMAM	20	$\simeq 0$
				MAD	25	2.094

Tabela 3: Anomalias do TPC-C por tipo.

	#Leituras Sujas	#Escritas Sujas	#Atualizações Perdidas	#Atrasos nas Escritas	#Atrasos nas Leituras	#Outros	#Total
TPC-C mono	0	0	0	0	0	0	0
TPC-C decomposição-1	0	13	0	0	14	41	68
TPC-C decomposição-2	0	4	0	0	11	10	25

exista uma operação a mais na execução considerada, então essa execução não será idêntica à do nosso padrão de leitura suja e por isso irá para o tipo "Outro".

4.3 Discussão

Ao contrário do que acontece com a CMMAM, o tempo de execução do MAD cresce de forma exponencial com o número de transações. Isto pode implicar que para aplicações com um número elevado de transações não seja possível obter um resultado em tempo útil. Nestes casos, existem duas alternativas: i) utilizar uma ferramenta com uma abordagem heurística para obter uma estimativa de quais são os microsserviços mais problemáticos, e de seguida aplicar a ferramenta MAD sobre esse conjunto de microsserviços; ii) gerar vários sub-conjuntos dos microsserviços da aplicação e processar cada um deles à vez, dado que a análise destes irá devolver uma resposta num menor intervalo de tempo e a junção dos resultados permite fazer a análise da decomposição de monólito para microsserviços. Esta limitação em termos da complexidade temporal é uma das desvantagens de utilizarmos o Z3 para encontrar as anomalias, no entanto contrabalança com o facto deste solucionador ser capaz de gerar todas as combinações de transações cujas execuções possuem anomalias com base na fórmula SMT gerada.

5 Conclusão

Este artigo apresentou a ferramenta MAD, desenvolvida para detetar as anomalias que podem surgir quando se decompõem as transações de um monólito para se executarem em microsserviços. Aplicámos a ferramenta a aplicações distintas, mostrando que esta consegue ser completa e precisa na deteção de anomalias e que pode ajudar no processo de escolha da melhor decomposição para sistemas como o TPC-C.

Agradecimentos: Este trabalho foi suportado pela FCT – Fundação para a Ciência e a Tecnologia, através dos projectos UIDB/50021/2020 e DACOMICO (financiado pelo OE com a ref. PTDC/CCI-COM/2156/2021).

Referências

1. Thones, J.: Microservices. IEEE Softw. **32**(1) (jan 2015) 116
2. Richardson, C.: Microservices Patterns: With examples in Java. (2018)
3. Papadimitriou, C.: The serializability of concurrent database updates. J. ACM **26**(4) (October 1979) 631–653
4. De Moura, L., Bjørner, N.: Z3: An efficient smt solver. In: TACAS'08, Budapest, Hungary. TACAS'08 (April 2008)
5. Santos, N., Silva, A.: A complexity metric for microservices architecture migration. In: ICSA'20, Salvador, Brazil. (March 2020)
6. Almeida, J., Silva, A.: Monolith migration complexity tuning through the application of microservices patterns. In: ECSA'20, L'Aquila, Italy. (September 2020)
7. Tan, C., Zhao, C., Mu, S., Walfish, M.: Cobra: Making transactional key-value stores verifiably serializable. In: OSDI'20, Virtual Event. (November 2020)
8. Biswas, R., Kakwani, D., Vedurada, J., Enea, C., Lal, A.: Monkeydb: Effectively testing correctness under weak isolation levels. Proc. ACM Program. Lang. **5**(OOPSLA) (October 2021)
9. Nagar, K., Jagannathan, S.: Automated detection of serializability violations under weak consistency. In Schewe, S., Zhang, L., eds.: CONCUR'18, Beijing, China. (September 2018)
10. Rahmani, K., Nagar, K., Delaware, B., Jagannathan, S.: Clotho: Directed test generation for weakly consistent database systems. Proc. ACM Program. Lang. **3**(OOPSLA) (October 2019)
11. Santos, M.: Microservice decomposition for transactional causal consistent platforms. Master's thesis, Instituto Superior Técnico, Universidade de Lisboa (June 2022)
12. Adya, A., Liskov, B., O'Neil, P.: Generalized isolation level definitions. In: ICDE'00, San Diego (CA), USA, USA, IEEE Computer Society (2000)
13. Nunes, L., Santos, N., Silva, A.: From a monolith to a microservices architecture: An approach based on transactional contexts. In: Software Architecture: 13th European Conference, ECSA 2019, Paris, France, September 9–13, 2019, Proceedings, Paris, France, Springer-Verlag (2019) 37–52
14. Kalia, A., Xiao, J., Krishna, R., Sinha, S., Vukovic, M., Banerjee, D.: Mono2micro: A practical and effective tool for decomposing monolithic java applications to microservices. In: Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering. ESEC/FSE 2021, Athens, Greece, Association for Computing Machinery (2021) 1214–1224
15. Brito, M., Cunha, J., Saraiva, J.a.: Identification of microservices from monolithic applications through topic modelling. In: Proceedings of the 36th Annual ACM Symposium on Applied Computing. SAC '21, New York, NY, USA, Association for Computing Machinery (2021) 1409–1418
16. Fowler, M.: Microservice trade-offs. `https://martinfowler.com/articles/microservice-trade-offs.html` Accessed: 25/05/2023.
17. Difallah, D., Pavlo, A., Curino, C., Cudre-Mauroux, P.: Oltp-bench: An extensible testbed for benchmarking relational databases. Proc. VLDB Endow. **7**(4) (December 2013) 277–288