Deteção de Violação de Invariantes em Microsserviços

João Fitas, Rafael Soares, António Rito Silva, Luís Rodrigues
{joao.quelhas.fitas, joao.rafael.pinto.soares, rito.silva, ler}@tecnico.ulisboa.pt

INESC-ID, Instituto Superior Técnico, Universidade de Lisboa

Resumo Num monólito, as funcionalidades de uma aplicação são executadas como transações, que estão isoladas umas das outras. Numa arquitetura de microsserviços, as funcionalidades podem ser compostas por múltiplas transações, cada uma executada num microsserviço diferente. Quando as funcionalidades são executadas concorrentemente, estas transações individuais podem intercalar-se de formas inesperadas, gerando estados globais que violem invariantes de correção. Este trabalho propõe uma ferramenta que permite detetar de forma automática execuções que causam violações de invariantes. A informação fornecida pela ferramenta ajuda os programadores a tomar medidas preventivas ou corretivas para evitar a ocorrência de anomalias ou para mitigar os seus impactos.

Palavras Chave — Microsserviços, Invariantes, Deteção de Anomalias

1 Introdução

Designam-se por aplicações de domínio complexo as aplicações que gerem múltiplas entidades, inter-relacionadas, e cuja integridade é expressa por condições não triviais sobre os estados destas entidades. Nestas aplicações, a execução concorrente de funcionalidades pode levar à quebra momentânea ou permanente da coerência do seu estado. Uma abordagem possível para gerir a complexidade destas aplicações é o *Domain Driven Design* [2] (DDD).

O DDD promove a organização das entidades em *agregados* [2,3,13], conjunto de entidades que são tipicamente acedidas recorrendo a transações e cuja coerência pode ser caracterizada de forma localizada. A coerência de cada agregado e a coerência mútua entre agregados pode ser formalizada através de invariantes [2,3,13], que a aplicação deve respeitar. Tipicamente, funcionalidades que acedem a vários agregados são codificadas como um grafo de transações sobre agregados, sendo que a execução concorrente de várias funcionalidades pode gerar intercalamentos que provocam a violação dos invariantes. Contudo nem todas as violações têm a mesma relevância para a coerência da aplicação: alguns invariantes podem ser relaxados durante a execução de uma funcionalidade e só precisarem de ser repostos antes de esta terminar, os quais denominamos de invariantes eventuais, enquanto outros deverão ser preservados em qualquer estado da aplicação, os quais denominamos de invariantes absolutos.

Um programador poderia ambicionar testar todas as execuções e combinações possíveis de funcionalidades de modo a identificar os cenários que resultem na violação de invariantes. No entanto, obter tal cobertura total da aplicação será impraticável, se não mesmo impossível, devido ao elevado número de combinações e execuções possiveis de uma dada aplicação. Algo que pode ser ultrapassado recorrendo a verificação formal em vez de execução de testes [1]. Neste trabalho propomos uma ferramenta de verificação formal de aplicações para detetar de forma automática intercalamentos que causem a violações de invariantes em aplicações de domínio complexo. A ferramenta foi concebida tendo em vista aplicações desenvolvidas para arquiteturas de microsserviços, recebendo o código fonte da aplicação, a especificação dos invariantes, a descrição das entidades dos agregados, e a descrição das funcionalidades. Utilizando esta informação, a ferramenta identifica quais os intercalamentos, e quais os valores de entrada, que levam à violação dos invariantes.

2 Exemplo

De modo a exemplificar a utilização da nossa ferramenta, iremos recorrer a uma aplicação de microsserviços simulando uma aplicação bancária simplificada. Esta aplicação é constituída por uma única entidade "Conta", constituída por um identificador de conta único e um saldo bancário. A aplicação aplica um invariante absoluto, garantindo que os saldos de todas as contas nunca devem ser negativos. A entidade "Conta" pode ser modificada recorrendo a uma funcionalidade "Levantamento", constituída por duas transações: uma transação denominada "Validação", que verifica se o levantamento pode ser efetuado sem quebrar o invariante, e uma transação "Levantamento" que subtrai o saldo da conta.

Devido à separação da funcionalidade em duas transações atómicas, é possível intercalar duas ou mais instâncias da funcionalidade "Levantamento" de modo a extrair mais saldo do que o existente na conta bancária, levando à violação do invariante da aplicação. No decorrer deste artigo, iremos demonstrar como a nossa ferramenta é capaz de detetar estas violações.

3 DAVIAC

Nesta secção, apresentamos uma ferramenta para a Deteção Automática de Violação de Invariantes em Aplicações de Domínio Complexo, que denominámos DAVIAC. De modo a detetar com precisão as possíveis quebras de invariantes de uma aplicação, a ferramenta codifica o código da aplicação e o seu conjunto de invariantes em fórmulas de *Teorias do Módulo da Satisfatibilidade* (do Inglês, *Satisfiability Modulo Theories*, SMT) [7]. Utilizando um solucionador SMT, a ferramenta explora o espaço de possíveis intercalamentos entre funcionalidades e argumentos iniciais de cada funcionalidade. Ao detetar uma quebra de invariante, a ferramenta regista o intercalamento e argumentos que levaram à sua quebra, apresentando todas as quebras detetadas no fim da sua execução.

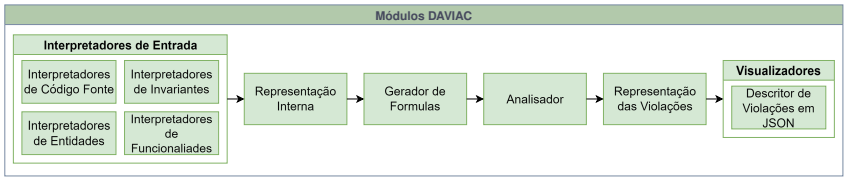


Figura 1: Sequência de módulos do DAVIAC

3.1 Arquitetura

A arquitetura e fluxo de execução do DAVIAC estão representados na Figura 1. De forma sucinta, a ferramenta segue a seguinte execução: primeiro, o código fonte, os invariantes, as entidades, e a informação sobre as funcionalidades e transações são fornecidos à ferramenta, onde são processados e transformados numa representação interna agnóstica à linguagem de programação usada na especificação fornecida. Esta representação interna é depois compilada para uma formulação SMT, sendo invocado o solucionador para detetar violações de invariantes. Finalmente, os intercalamentos que geram violações são agregados e processados por um visualizador, que apresenta estes resultados ao programador. De seguida, descrevemos cada um dos módulos do DAVIAC em mais pormenor.

3.2 Interpretadores de Entrada

O DAVIAC usa quatro classes de interpretadores: i) interpretadores de código fonte, ii) interpretadores de invariantes, iii) interpretadores de entidades, iv) interpretadores de funcionalidades.

Interpretadores de Entidades Os interpretadores de entidades são responsáveis pela tradução das entidades do sistema (incluindo os seus atributos e respetivos tipos) para a representação interna suportada pela ferramenta. Estas entidades são tradicionalmente representadas em SQL ou Mapeamento objeto-relacional (do Inglês, *Object–Relational Mapping*, ORM) [11]. Utilizando os esquemas de base de dados utilizados por ambas as técnicas, a ferramenta é capaz de extrair os atributos e tipos de cada entidade. Atualmente, o protótipo suporta apenas entidades em SQL, considerando cada tabela SQL uma entidade e as suas colunas constituintes os seus atributos. A informação sobre os esquemas de base de dados podem ser obtidas a partir dos comandos de criação de bases de dados SQL, quer sejam estes criados manualmente pelo programador ou gerados automaticamente pelo ORM. A Listagem 1.1 apresenta um exemplo da representação suportada atualmente pela ferramenta. Esta apresenta um ficheiro SQL com os comandos de criação de tabelas, onde existem os dados referentes à entidade `contas`, constituída por dois atributos, `id` e `saldos`, ambos do tipo inteiro. Atualmente, a ferramenta apenas suporta atributos com tipos básicos.

```
CREATE TABLE contas (
    id          INTEGER,
    saldo       INTEGER,
    PRIMARY KEY (id)
);
```

Listagem 1.1: Exemplo Ficheiro descritor de Entidades

```
{
    "Absolute Invariants": [ "contas.saldo >= 0" ],
    "Eventual Invariants": [ "WHERE contas.id == 1 THEN contas.saldo > 100" ]
}
```

Listagem 1.2: Exemplo Ficheiro descritor de Invariantes

Interpretadores de Invariantes Estes invariantes capturam as regras de coerência do domínio, que segundo o DDD devem ser formalizadas para cada agregado durante o seu desenho. Esta formalização é fornecida à ferramenta, incluindo informação relativamente ao tipo de cada invariante (i.e. eventual ou absoluto). Clarificando as especificações de cada invariante, um invariante eventual é violado caso este não se verifique num estado quisciente da aplicação. Ou seja, este pode ser violado no decorrer da execução de uma funcionalidade, desde que a sua correção seja reposta no fim da sua execução. Por outro lado, invariantes absolutos deverão ser mantidos em todos os pontos de execução da aplicação.

Os interpretadores de invariantes são responsáveis por traduzir esta especificação de alto nível para uma representação interna, relacionando cada invariante com as entidades e funcionalidades envolvidas.

Concretamente, o interpretador atual no protótipo recebe um ficheiro JSON que descreve os invariantes numa sintaxe semelhante a SQL, como representado na Listagem 1.2. Expandindo o código fonte anterior, o ficheiro apresenta duas invariantes: um invariante absoluto, indicando que em nenhum estado a propriedade `saldo` de qualquer entidade `contas` pode ser menor do que 0, e um invariante eventual, indicando que qualquer entidade `contas` cujo `id` seja 1 tem de ter saldo maior que 100 em qualquer estado onde nenhuma funcionalidade esteja a operar sobre esta entidade.

Atualmente, o protótipo suporta invariantes que afetam todas as entidades de um tipo ou um subconjunto de entidades restritas por uma dada condição, indicadas por uma cláusula WHERE. O protótipo suporta encadeamentos lógicos de condições com conjunções e disjunções e os seguintes operadores: "`<`", "`>`", "`<=`", "`>=`", "`==`" e "`!=`".

Interpretadores de Código das Transações Os interpretadores de código fonte das transações são responsáveis por identificar todos os acessos a entidades efetuados pela aplicação. Estes acessos são representados como um grafo de execução, contendo as operações executadas às entidades acedidas como também

```
1    public void saldo(int id_conta) throws SQLException {
2        PreparedStatement stmt1 = connection.prepareStatement("SELECT saldo
     FROM contas WHERE id = ?");
3        stmt1.setInt(1, id_conta);
4        ResultSet rs = stmt1.executeQuery();
5        rs.next();
6        int saldo = rs.getInt("saldo");
7    }
```

Listagem 1.3: Exemplo de ficheiro código fonte

as condições necessárias para a execução das operações. Este grafo é traduzido para uma representação interna agnóstica à linguagem de programação utilizada pela aplicação, permitindo a implementação de vários interpretadores de código adaptados às características do código fonte. A especificação desta representação interna é descrita em mais pormenor na Secção 3.3.

O protótipo atual inclui um interpretador para código Java que utilize expressões SQL como método de manipulação das entidades. Este interpretador utiliza a biblioteca JavaParser[1] para construir e percorrer uma *Árvore Sintática Abstrata* (do inglês, *Abstract Syntax Tree*, AST) que representa as transações que compõem as funcionalidades da aplicação sob teste. Ao percorrer esta árvore, o interpretador gera os nós correspondentes na representação interna, que retém os tipos das entidades no código, quais os argumentos de cada transação e todas as expressões que sejam usadas em condições ou escritas para entidades. A ferramenta poderá ser estendida no futuro para suportar aplicações que utilizem Spring[2], DJango[3], ou outras *molduras* (do Inglês, *frameworks*) de programação.

A Listagem 1.3 apresenta como exemplo código Java suportado pelo DAVIAC, nomeadamente um método "saldo" de uma aplicação bancária simples.

Interpretadores de Funcionalidades Os interpretadores de funcionalidades são responsáveis por traduzir o encadeamento de transações em funcionalidades. Uma funcionalidade corresponde a um encadeamento de transações executados em um ou mais microsserviços e por uma ordem definida pela funcionalidade. A informação relativamente a estes encadeamentos e ordens permitem restringir o espaço de procura da ferramenta, evitando explorar intercalamentos de transações impossíveis de se executar pela aplicação.

Esta informação é obtida a partir dos documentos de desenho da aplicação seguindo DDD. Na ausência destes documentos, o programador poderá extrair esta informação a partir do código fonte. Por exemplo, no caso de existir um coordenador central de funcionalidades [4], esta informação poderá ser extraída deste. No caso de sistemas baseados na publicação e subscrição de eventos, a informação pode ser derivada a partir dos registos de acesso à fila de eventos, gerando um grafo de execução das funcionalidades.

[1] https://javaparser.org/
[2] https://spring.io/
[3] https://www.djangoproject.com/

```
{
    "Levantamento": [
        "saldo",
        "levantar"
    ]
}
```

Listagem 1.4: Exemplo Ficheiro descritor de Funcionalidades

A Listagem 1.4 apresenta um exemplo da informação atualmente utilizada pelo protótipo da ferramenta. Esta recebe um ficheiro JSON composto pelas funcionalidades, as suas transações e a sua ordem de execução respetiva. Podemos observar uma única funcionalidade, `Levantamento`, constituída por duas transações, `saldo` e `levantar`, executadas por esta ordem respetiva.

3.3 Representação Interna

A representação interna permite desacoplar a geração de fórmulas SMT da codificação usada na descrição do sistema, facilitando a expansão da ferramenta. É mantida na representação interna a informação extraída dos interpretadores de entrada necessária para a formulação SAT do DAVIAC, sendo esta constituída por três listas de objetos: entidades, invariantes e funcionalidades. Entidades mantêm informação relativamente aos seus atributos, especialmente quanto ao tipo destes. Invariantes definem as restrições impostas aos possíveis estados coerentes de atributos de entidades, sendo categorizados como absolutos ou eventuais. Funcionalidades mantêm uma sequência de transações, sendo cada transação representada por uma AST, capturando as entidades e atributos manipuladas por cada transação. Particularmente, a representação em ASTs é uma representação agnóstica à linguagem de programação utilizada pela aplicação, permitindo reter os encadeamentos de acessos a entidades tal como as condições necessárias à execução dos mesmos.

A representação interna permite estabelecer ligações entre os invariantes, os atributos de entidades relevantes para a correção do invariante, e as transações que interagem com estes atributos. Estas ligações serão fulcrais para a formulação SAT, permitindo restringir o espaço de procura para a validação da correção dos invariantes apenas as funcionalidades e transações relevantes para o invariante.

A Figura 2 apresenta a representação interna do exemplo recorrente deste artigo, dando destaque às ligações estabelecidas relativamente ao invariante absoluto "`Contas.saldo >= 0`". Neste exemplo, o invariante absoluto restringe a gama de valores possíveis para o atributo "saldo" da entidade "Contas", sendo este atributo manipulado pelas transações "saldo" e "levantar" da funcionalidade "Levantamento". Assim, a ferramenta é capaz de identificar as transações relevantes para um dado invariante, restringindo a procura de execuções que violem o invariante apenas a execuções que incluam estas transações.

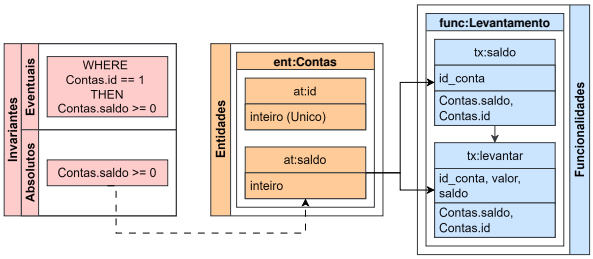


Figura 2: Exemplo Representação interna de Aplicação no DAVIAC

3.4 Gerador de Fórmulas

Este componente é responsável por compilar a representação interna para SMT, gerando as fórmulas que serão verificadas pelo analisador para descobrir violações. A ferramenta irá explorar, para cada invariante fornecido, todas as execuções possíveis de funcionalidades e transações que interajam sobre as entidades relevantes para o invariante. Para tal, a ferramenta explora todos os intercalamentos possíveis entre funcionalidades, não só entre funcionalidades diferentes como também instâncias concorrentes da mesma funcionalidade. Além disso, para cada intercalamento, a ferramenta cobre todos os valores de entrada para cada transação e estados iniciais da aplicação possíveis. A ferramenta apenas explora estados iniciais que garantam inicialmente a correção de todos os invariantes da aplicação. Utilizando a informação extraída pela AST, a ferramenta é capaz de explorar todas as ramificações de acessos a entidades geradas a partir de acessos condicionados de cada transação.

Alavancando o facto que a procura de violações é independente para cada invariante, o DAVIAC gera uma fórmula SAT para cada invariante do sistema. Esta separação permite não só a paralelização da procura de violações, permitindo invocar execuções concorrentes do solucionador SAT para cada fórmula gerada, como também reduz a complexidade da formulação SAT, melhorando o desempenho da procura.

De modo a restringir a profundidade máxima da procura da ferramenta, o número de instâncias de funcionalidades envolvidas numa dada execução é restringido a um valor máximo t_{max}, definido na configuração inicial da ferramenta. No protótipo atual, t_{max} é definido pelo número máximo de funcionalidades que afetam o invariante. Iremos explorar em mais detalhes o impacto da escolha deste valor na precisão da ferramenta na Secção 3.8.

3.5 Analisador

Este componente verifica a satisfatibilidade das fórmulas produzidas pelo gerador. Utilizamos o Z3[4] como solucionador, devido à sua popularidade na área e

[4] https://github.com/Z3Prover/z3

eficiência. O analisador é invocado para cada fórmula gerada, e devido à independência entre fórmulas, várias instâncias paralelas de Z3 podem ser invocadas para acelerar a análise, reduzindo o tempo de execução linearmente com o aumento do número de instâncias de Z3 disponíveis. No caso de uma fórmula ser satisfazível, o modelo de saída é usado para gerar uma representação da violação. Caso contrário, é garantido que a execução representada não poderá gerar uma violação de invariante, podendo ser descartada. Concretamente, a aplicação de um solucionador SAT fornece a garantia de cobertura total da aplicação até um dado t_{max}, dado que o DAVIAC analisa todas as execuções para todas as combinações possíveis de funcionalidades de cumprimento até t_{max}. Caso a ferramenta não descubra nenhuma violação de invariante, temos a garantia de que para execuções concorrentes de até t_{max} funcionalidades, não é possível que ocorram violações de invariantes.

3.6 Representação da Violação de Invariantes

Esta representação é gerada extraindo informação do modelo de saída do analisador, contendo os dados da violação descoberta. Nomeadamente, é representado o invariante que foi violado, as funcionalidades envolvidas e qual a ordem de execução das transações, incluindo os estados iniciais e valores de entradas que levaram à violação. Esta representação tem como objetivo separar o resultado da inferência da sua apresentação ao programador, permitindo que a violação seja reportada ao programador de várias formas, permitindo aos programadores criarem vários visualizadores adaptados aos seus casos de uso.

3.7 Visualizadores

Estes componentes são responsáveis por apresentar as violações de invariantes ao programador. A ferramenta está preparada para utilizar vários visualizadores. No protótipo atual, existe apenas um visualizador, apresentando uma descrição das execuções que geram as violações em JSON.

Este visualizador gera um ficheiro JSON que contém uma descrição temporal da execução que levou à violação do invariante. Esta descrição inclui a lista de estados das entidades que compõem a aplicação intercalados com as funcionalidades que executaram originando cada estado, bem como os seus argumentos. Futuramente pretendemos substituir este visualizador por um gerador de grafos, como o presente na Figura 3, onde temos a sequência de estados (a verde) intercalados com as funcionalidades (a azul). Neste exemplo, o intercalamento de duas instâncias da funcionalidade `Levantamento` leva a que, no `Estado 2`, a conta com `id` 0 tenha saldo negativo, quebrando o invariante.

3.8 Limitações

Escolha de t_{max} De modo a limitar a procura de execuções de instâncias concorrentes sobre as mesmas funcionalidades, limitamos o número de instâncias

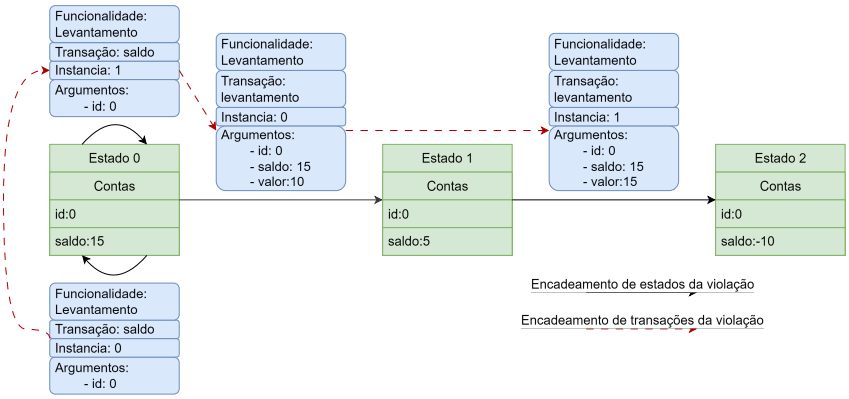


Figura 3: Exemplo de violação de um invariante

```
//Invariante: Conta.saldo < 100

func1_T0(){
    assert Conta.saldo < 50
}

func1_T1(){
    Conta.saldo += 10
}
```

Listagem 1.5: Exemplo de limitação da ferramenta

a serem exploradas pela formulação SAT a um valor t_{max}. Por omissão, este valor é definido relativamente ao número de funcionalidades que manipulam cada invariante, de modo a que a ferramenta explore intercalamentos que envolvam pelo menos uma instância de todas as funcionalidades que afetam um invariante. Contudo, é concebível que existam violações de invariantes que necessitem de um número superior de instâncias concorrentes de modo a serem detetadas.

Considere uma aplicação simples, ilustrada na Listagem 1.5, com apenas uma funcionalidade `func1`, consistida por duas transações: `T0`, que verifica que um atributo atributo `saldo` de uma entidade `Conta` é menor que 50 recorrendo a um `assert`, terminando a funcionalidade se a restrição no `saldo` não se verificar, e `T1`, que incrementa o atributo `saldo` por 10 caso `T0` seja validada com sucesso. Nesta aplicação, existe um invariante sobre o atributo `saldo`, querendo garantir-se que o `saldo` nunca é superior ou igual a 100. Neste exemplo, com apenas uma invocação da `func1` é impossível violar o invariante, pois só é efetuado um incremento ao atributo `saldo` caso este seja menor que 50. No entanto, considerando um caso em que o estado inicial do atributo `saldo` encontra-se a 49 e são executadas 6 ou mais instâncias da funcionalidade concorrentemente, é possível encontrar uma execução onde todas as instâncias concorrentes executam `T0` primeiro, validando que o `saldo` encontra-se menor que 50, e de seguida todas

executam `T1`, incrementando o `saldo` por 60 ou mais, violando assim o invariante. Consideramos como trabalho futuro extrapolar o t_{max} necessário para garantir a deteção destes casos de violações.

Deteção de invariantes únicos Durante a procura de violações de invariantes, a mesma violação poderá ser encontrada sobre várias execuções diferentes. Atualmente, a ferramenta apresenta todas as execuções detetadas que levaram à violações de invariantes, podendo apresentar várias versões da mesma violação. No futuro, pretendemos agrupar as execuções onde a mesma violação é encontra, de forma a apresentar ao programador uma vista mais informativa para cada violação que fornece vários exemplos de como uma violação pode surgir.

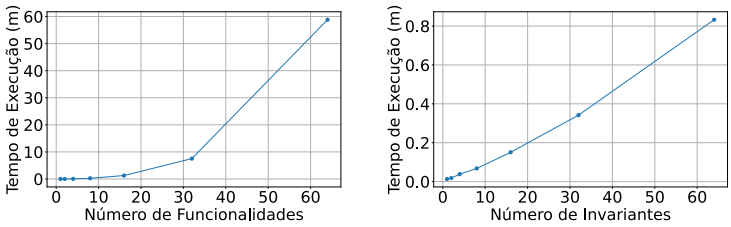
4 Avaliação

Para avaliar a capacidade do DAVIAC detetar violações de invariantes relevantes em aplicações reais, adaptamos a versão de microsserviços da aplicação Quizzes Tutor[5] modelada em [9] para a sintaxe suportada pelo protótipo da ferramenta. Esta é composta por quatro funcionalidades e quatro invariantes (2 absolutos e 2 eventuais), onde, em média, cada invariante é acedido, para leitura ou escrita, por três funcionalidades. Limitando o numero de execuções concorrentes de funcionalidades exploradas pela ferramenta (t_{max}) a três funcionalidades, a análise desta aplicação demora aproximadamente um minuto e revela 3413 violações de invariantes. Este resultado confirma a capacidade do DAVIAC em analisar uma aplicação próxima de uma aplicação real em tempo útil. Parte significativa deste número de violações representam a mesma violação detetada em várias execuções diferentes. Especialmente, grande número de violações detetadas são execuções causadas por apenas duas funcionalidades, detetadas novamente quando intercaladas com funcionalidades adicionais.

Para além da capacidade de produzir resultados relevantes, o tempo da análise é crítico para uma ferramenta de teste. De modo a avaliar a capacidade de escala do DAVIAC, medimos o tempo de execução do DAVIAC para uma aplicação sintética, variando o número de invariantes e funcionalidades da aplicação. Para estas experiências, t_{max} é fixado a duas funcionalidades. A aplicação sintética inicial é constituída por um invariante absoluto que relaciona dois atributos e uma funcionalidade, constituída por duas transações, em que cada uma atualiza um dos atributos do invariante.

Começamos por medir o impacto que o número de funcionalidades que interagem com um invariante tem sobre o desempenho. Para tal, aumentamos o número de funcionalidades da aplicação sintética inicial que interagem com o mesmo único invariante, introduzindo funcionalidades adicionais com o mesmo conteúdo que a original. Podemos observar os resultados desta experiência na Figura 4a, onde medimos o tempo de execução da ferramenta em relação ao

[5] https://quizzes-tutor.tecnico.ulisboa.pt/



(a) Vs. funcionalidades (b) Vs. invariantes

Figura 4: Tempo de execução em função do número de funcionalidades e de invariantes.

número de funcionalidades, aumentando o número de funcionalidades da aplicação até 64. Como esperado, o tempo de execução apresenta um crescimento aproximadamente exponencial, devido ao aumento de combinações possíveis de funcionalidades e execuções que a ferramenta terá de validar.

De seguida, medimos o impacto que o número de invariantes de uma aplicação tem sobre o desempenho. Para tal, variamos o número de invariantes existentes na aplicação, mantendo um número de funcionalidades que interagem com cada invariante constante. Como esperado, como se pode observer na Figura 4b, o desempenho da ferramenta cresce linearmente com o número de invariantes, dado que a análise de violações de cada invariante é independente. No futuro, esta análise poderá também ser feita em paralelo, reduzindo drasticamente o tempo de execução.

Finalmente, procuramos avaliar o impacto no tempo de execução de variar o valor de t_{max}. Para o efeito, consideramos uma aplicação constituída por quatro funcionalidades, sendo que todas afetam um único invariante. Como esperado, , como se pode observer na Figura 5, o tempo de execução apresenta um crescimento aproximadamente fatorial devido ao aumento de combinações possíveis de funcionalidades e execuções que a ferramenta terá de validar.

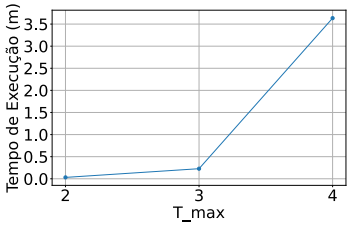


Figura 5: Tempo de execução em função do t_{max}.

5 Trabalho Relacionado

Semelhante ao DAVIAC, vários trabalhos do estado da arte focam-se na validação da correção de aplicações. Nesta secção, iremos explorar alguns dos trabalhos que mais influenciaram o desenvolvimento do DAVIAC.

O Transactional Causal Consistent Simulator [9] simula a semântica transacional de microsserviços num ambiente centralizado. Este simulador é capaz de manipular a propagação de dados entre microsserviços, permitindo simular e manipular o intercalamento entre funcionalidades de uma forma precisa. Embora esta ferramenta tenha como foco simular transações com semânticas de coerência causal transacional [5], esta permite aos programadores criarem casos de testes mais precisos, capazes de verificar a correção dos invariantes da aplicação. No entanto, ao contrário do DAVIAC, os programadores são responsáveis por criar estes casos de testes, que não só é um processo demorado, como também não oferece garantias relativamente à completude da verificação da aplicação.

À semelhança do DAVIAC, vários trabalhos no estado da arte utilizam SMT de modo a modular e verificar a correção da aplicação. Um sub-conjunto de trabalhos de interesse focam-se na verificação de anomalias transacionais em aplicações, utilizando formulações SAT para modular a aplicação e verificar todos os intercalamentos possíveis entre transações [8,10,12,6]. Embora nenhuma destas ferramentas seja capaz de verificar a existência de violações de invariantes, a sua modelação de aplicações em fórmulas SAT foi usada como inspiração para a formulação utilizada pelo DAVIAC. Mais especificamente, inspirámos-nos na formulação do Noctua [6], cuja formulação simples da aplicação permite um melhor desempenho que o estado da arte. Para o melhor do nosso conhecimento, o DAVIAC é a primeira ferramenta capaz de verificar a existência de violações de invariantes em ambientes microsserviços.

6 Conclusões e Trabalho Futuro

Neste artigo descrevemos uma nova ferramenta, o DAVIAC, que permite detetar violações de invariantes na execução de funcionalidades em aplicações de microsserviços. Recorrendo a formulação SAT, o DAVIAC foi capaz de detetar violações de invariantes numa aplicação representativa de microsserviços em tempo útil, apresentando as execuções exatas que geraram cada violação. No futuro, pensamos em expandir o DAVIAC para suportar níveis de coerência diferentes para cada transação, adicionar a opção de permitir execução de funcionalidades com encadeamentos mais complexos, e ainda expandir as tecnologias de entrada suportadas nativamente assim como oferecer mais visualizadores de saída.

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. Beyer, D., Lemberger, T.: Software verification: Testing vs. model checking - A comparative evaluation of the state of the art. In: HVC'17. Haifa, Israel (Nov 2017)
2. Evans, E.: Domain-driven design: tackling complexity in the heart of software. Addison-Wesley Professional (2004)
3. Evans, E.: Domain-driven design reference: Definitions and pattern summaries. Dog Ear Publishing (2014)
4. Laigner, R., Zhou, Y., Salles, M., Liu, Y., Kalinowski, M.: Data management in microservices: State of the practice, challenges, and research directions. VLDB **14**(13) (Sep 2021)
5. Lykhenko, T., Soares, R., Rodrigues, L.: Faastcc: efficient transactional causal consistency for serverless computing. In: Middleware '21. Online (Québec City, Canada) (Dec 2021)
6. Ma, K., Li, C., Zhu, E., Chen, R., Yan, F., Chen, K.: Noctua: Towards automated and practical fine-grained consistency analysis. In: EuroSys '24. Athens, Greece (Apr 2024)
7. Moura, L., Bjørner, N.: Z3: An efficient smt solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS '08. Budapest, Hungary (Apr 2008)
8. Nagar, K., Jagannathan, S.: Automated detection of serializability violations under weak consistency. In: CONCUR'18. Beijing, China (Sep 2018)
9. Pereira, P., Silva, A.R.: Transactional causal consistent microservices simulator. In: DAIS'23. Lisbon, Portugal (Jun 2023)
10. Rahmani, K., Nagar, K., Delaware, B., Jagannathan, S.: CLOTHO: directed test generation for weakly consistent database systems. OOPSLA'19 (Oct 2019)
11. Rogers, S.: The pros and cons of object relational mapping (orm) (2019), https://midnite.uk/blog/the-pros-and-cons-of-object-relational-mapping-orm, accessed: 2024-06-25
12. Romão, V., Soares, R., Manquinho, V., Rodrigues, L.: Deteção automática de anomalias em arquiteturas de microsserviços. In: Inforum'23. Porto, Portugal (Sep 2023)
13. Vernon, V.: Implementing domain-driven design. Addison-Wesley (2013)