Aplicação do Fecho de Programas na Deteção de Anomalias de Concorrência

Diogo G. Sousa, João M. Lourenço, Eitan Farchi, and Itai Segall*

CITI — Departamento de Informática,
Universidade Nova de Lisboa, Portugal
dm.sousa@campus.fct.unl.pt joao.lourenco@fct.unl.pt
farchi@il.ibm.com itais@il.ibm.com

Resumo Uma das estratégias para tirar partido dos múltiplos processadores disponíveis nos computadores atuais passa por adaptar código legado, inicialmente concebido para ser executado num contexto meramente sequencial, para ser agora executado num contexto *multi-threading*. Nesse processo de adaptação é necessário proteger apropriadamente os dados que são agora partilhados e acedidos por diferentes *threads* concorrentes. A proteção dos dados com *locks* e usando uma granulosidade grossa inibe a concorrência e opõe-se ao objetivo inicial de explorar o paralelismo suportado por múltiplos processadores. Por outro lado, a utilização de uma granulosidade fina pode levar à ocorrência de anomalias próprias da concorrência, como *deadlocks* e violações de atomicidade (*high-level data races*). Este artigo discute o conceito de *fecho de um programa* e uma metodologia que, quando aplicados em conjunto, permitem adaptar código legado para o tornar *thread-safe*, garantindo a ausência de violações de atomicidade na versão corrente do software e antecipando algumas violações de atomicidade que poderão ocorrer em versões futuras do mesmo software.

Keywords: Violações de Atomicidade; *High-Level Data Races*; *Thread Safety*; Análise Estática; Verificação; Concorrência

1 Introdução

Com o advento de processadores *multicore*, a exploração da concorrência tornou-se um requisito primário no processo de desenvolvimento de software. Programas inicialmente escritos para um modelo de execução sequencial necessitam de ser adaptados para suportar concorrência e tirar partido dos vários processadores agora disponíveis. Esta tarefa de adaptação é propensa a erros pois, para identificar e proteger regiões críticas, o programador tem de ter em consideração não apenas quais são os dados partilhados, mas também a ordem pela qual estes

* Este trabalho foi parcialmente financiado pela União Europeia, no contexto da COST Action IC1001 (Euro-TM), e pela Fundação para a Ciência e Tecnologia (FCT/MCTES), no contexto dos projetos de investigação PTDC/EIA-EIA/108963/2008 e PTDC/EIA-EIA/113613/2009 e bolsa de investigação SFRH/BD/41765/2007.

deverão ser processados pelos vários métodos. Se essa ordem não for respeitada, o programa poderá apresentar comportamentos erróneos nem sempre fácil de diagnosticar, como é o caso das *high level data races*. Intuitivamente uma *high level data race* (HLDR) representa uma anomalia causada por uma sequência de acessos atómicos a variáveis que, devido às dependências entre essas variáveis, deveriam acontecer num único acesso atómico. Este problema é particularmente relevante quando o código que é executado pelos múltiplos *threads* concorrentes está a invocar serviços de uma biblioteca ou de um módulo de software desenvolvido por terceiros, pois a combinatória das invocações concorrentes a métodos da API pode ser muito grande e muitas dessas combinatórias podem violar as restrições impostas pela dependência de dados entre os métodos da API que, frequentemente, não estão documentadas. A Listagem 1.1 ilustra um caso de uma HLDR na definição da operação `mover`, onde se assumiu que não necessitava de ser declarada como `@atomic` porque os dois métodos que esta utiliza já são atómicos.

Listagem 1.1: Exemplo de uma *high level data race*.

```
@Atomic
boolean remover (Set s, int i) {
    // remove 'i' de 's'
    // retorna true sse 'i' existe em 's'
    ...
}

@Atomic
void inserir (Set s, int i) {
    // insere 'i' em 's'
    ...
}

void mover (Set a, Set b, int i) {
    // move 'i' de 'b' para 'a'
    if (remover (b, i))
        inserir (a, i);
}
```

Este exemplo, pela sua simplicidade, torna imediato que o método `mover` também deveria estar definido como um bloco atómico. No entanto este facto não pode ser generalizado e, em programas maiores e mais complexos, frequentemente não é óbvio se um bloco composto por sub-blocos atómicos deve também ser ou não atómico. Uma estratégia pessimista de sobre-especificação de blocos atómicos, agrupando desnecessariamente conjuntos de operações em blocos atómicos, não compromete a semântica nem a correção do programa, mas limita a concorrência no programa e, portanto, reduz o coeficiente de utilização dos recursos hardware, em particular dos processadores.

Este artigo apresenta uma metodologia baseada em análise estática de programas em Java que possibilita a identificação de possíveis cenários de sub-

especificação de blocos atómicos. A execução concorrente destes blocos, num ambiente de *multi-threading*, poderá comprometer a correção do programa. A aplicação da metodologia proposta permite ao programador corrigir os erros causados por HLDR na versão corrente do programa, bem como identificar e documentar as restrições na sequência de invocação de serviços da API que, se não forem cumpridas, podem originar HLDR em futuras versões desse software.

Na próxima seção iremos apresentar a metodologia geral para transformação de uma classe Java, inicialmente desenvolvida para ser utilizada num contexto sequencial, de forma a torná-la passível de utilização num contexto de *multi-threading*. Na Seção 3 iremos apresentar o conceito de *fecho de um programa*, acompanhado por alguns exemplos da sua utilização. Na Seção 4 apresentamos em detalhe a metodologia a seguir para deteção de *high level data races* introduzidas pela possível utilização concorrente de métodos públicos da classe. Segue-se a validação da abordagem na Seção 5, a apresentação do trabalho relacionado na Seção 6, terminado na Seção 7 com as conclusões.

2 Metodologia Geral

Uma *API* de uma classe é composta pelo conjunto de métodos públicos que esta expõe ao resto do programa. Assumindo que os métodos podem criar "internamente" novos *threads*, é possível que existam *data races* e *deadlocks* mesmo sem considerar as interações entre os métodos da API. A análise apresentada deteta HLDR tanto nos métodos da API individualmente, como nas possíveis utilizações dos objetos dessa classe por parte de um programa cliente *multi-threaded*.

Neste artigo assumimos que as classes usam memória transacional como mecanismo de controlo de concorrência. múltiplas vantagens na especificação da concorrência num programa. Ao invés de usar *locks*, que garantem exclusão mutua, as transações em memória tipicamente executam de forma otimista, em concorrência real, assumindo que não vão existir interferências entre transações. A infraestrutura de execução que suporta o sistema transacional monitoriza os acessos à memória, aborta as transações em caso de conflito, repões a memória no estado em que estava no início da transação, e reinicia a transação abortada.

Utilizar num contexto *multi-threading* uma classe que, inicialmente, foi desenvolvida para um contexto de execução sequencial, requer um processo de transformação para a tornar *thread-safe*. Para isso, é necessário eliminar os *[low level] data races* através da proteção apropriada das regiões críticas do programa usando blocos atómicos. De forma a maximizar a concorrência, propomos a utilização de blocos atómicos de grão fino, com o escopo tão pequeno quanto possível. As potenciais anomalias introduzidas pela violação das restrições de dependência entre os blocos atómicos, que irão conduzir a HLDR, serão identificadas pela metodologia apresentada e poderão consequentemente ser corrigidas pelo programador.

A metodologia a seguir para transformar uma implementação de uma API (de uma biblioteca ou módulo de um programa) segue os seguintes passos:

1. Utilizar uma ferramenta apropriada para identificar todos os dados (variáveis) partilhados;
2. Proteger os acessos aos dados partilhados agregando-os no interior de um bloco atómico tão pequeno quanto possível;
3. Utilizar um detetor de HLDR, e.g., MoTH [4], para identificar as HLDR existentes nos métodos da API, caso estes sejam internamente *multi-thread*;
4. Utilizar o conceito de Fecho de Programas (apresentado na secção seguinte) para identificar as HLDR existentes entre métodos da API;
5. Utilizar ainda o conceito Fecho de Programas para identificar e documentar todas as restrições na utilização dos métodos da API que, se não respeitadas em versões futuras do software, levarão à introdução de novas HLDR no programa.

Ao longo deste artigo assumiremos um conjunto de pressupostos para a metodologia proposta, nomeadamente:

a) As zonas críticas da classe estão protegidas por memória transacional, pelo que o programa está isento de *[low level] data races*. Há vários pacotes de software que podem ajudar o programador a identificar *[low level] data races* num programa Java, ajudando portanto na localização dos blocos de código que necessitam de ser protegidos como uma região crítica (bloco atómico).
b) Todo o acesso aos atributos dos objetos é feito através da respetiva API. As boas metodologias de programação sugerem que os acessos aos atributos dos objetos sejam feitos através de *getters* e *setters*. O *overhead* associado à invocação destes *getters* e *setters* pode ser reduzido ou totalmente eliminado através de uma otimização automática que o transforma o código dessas invocações num acesso direto aos atributos em causa.
c) Os blocos transacionais não contêm operações irreversíveis, ou o sistema de transacional entra em modo pessimista [3] na sua presença. A execução de operações irreversíveis dentro transações é problemática caso a transação necessite de abortar. Uma aproximação que resolve esta limitação é assegurar que não há outras transações a executar em concorrência com a transação que irá executar operações irreversíveis. Desta forma a transação irá sempre ter sucesso, nunca sendo portanto necessário reverter as operações.
d) O conjunto de variáveis partilhadas acedidas em cada transação é conhecido. Esta informação pode com alguma facilidade ser obtida através de um processo de análise estática do código Java ou do Bytecode gerado pelo compilador.
e) A ordem pela qual as transações poderão ser executadas é também conhecida. Obter esta informação com precisão total é muito difícil apenas por análise estática, mas através da utilização de técnicas como *happens-before* e *may-run-in-parallel*, é possível conseguir aproximações razoáveis que permitem satisfazer este pressuposto.

3 Fecho de um Programa

Dada uma transação, a sua *view* é o conjunto de variáveis partilhadas a que esta acede. O conjunto das *views* de um *thread* t, denotado por $V(t)$, corresponde ao conjunto das *views* das transações executadas por t. Uma *view maximal* de um *thread* t é uma *view* de t que não é subconjunto de nenhuma outra *view* de t. O conjunto das *views maximais* de um *thread* t_i é denotado por $V_M(t_i)$.

Definimos agora a caracterização de HLDR, baseada na definição originalmente introduzida em [1].

Definição 1 (*High Level Data Race*) *Seja*

$$\varphi(t, v_m) = \{v_m \cap v \mid v \in t \ \wedge \ v \cap v_m \neq \emptyset\},$$

então é possível ocorrer uma HLDR entre dois threads t_A *e* t_B *se existir alguma view máxima* $v_m \in V_M(t_A)$ *tal que* $\varphi(t_B, v_m)$ *não forma uma cadeia. Um conjunto de views* $\mathcal{V}$ *forma uma cadeia quando* $\forall_{u,v \in \mathcal{V}}\ u \subseteq v \vee u \supseteq v$*, ou seja, se o conjunto* $\mathcal{V}$ *é totalmente ordenado sobre* $\subseteq$*.*

Definimos agora o grafo direcionado de dependências de um *thread* t, i.e., o grafo das *views* de t que estão relacionadas entre si através de uma relação de dependência.

Definição 2 (Grafo de Dependências de um *Thread*) *O grafo de dependências de um thread* t*,* $G_t = (V_t, E_t)$*, é definido por:*

$$V_t = V(t)$$
$$E_t = \{(u, v) \in V_t^2 \mid \delta(u, v)\}$$

em que $\delta(u, v)$ *é o predicado de dependência entre views, que é parâmetro da análise.*

Ao longo deste artigo iremos assumir que o predicado δ é definido como $\delta(u, v) \Leftrightarrow u \prec v \ \wedge \ u \cap v \neq \emptyset$, onde a expressão $u \prec v$ denota que a execução da transação correspondente à *view* u pode correr antes da transação correspondente à *view* v. Esta predicado pode ser computado de forma eficiente, tendo também em conta o controlo de fluxo do programa, e captura as dependências que as transações têm entre si através de variáveis comuns. Isto refina a análise pois só considera as transações que foram executadas em sequência por cada *thread*. Este refinamento reduz a probabilidade de falso positivos, pois descarta cenários de execução que não são possíveis no programa. No entanto, é possível utilizar outras definições alternativas para o predicado $\delta(u, v)$ que melhor se adequem a um programa, módulo ou classe específicos, de forma a minimizar o número de falsos positivos.

A noção de *view de fecho* de um *thread* reflete um conjunto de variáveis cujo acesso deveria, potencialmente, ser sempre realizado num único bloco atómico. Esta definição captura a dependência transitiva entre transações do grafo de dependências de um *thread*.

Definição 3 (*View de Fecho* de um *Thread*) *Seja t um thread e $p = \langle v_1, \cdots, v_n \rangle$ um caminho maximal e simples em G_t, então diz-se que $C_p = v_1 \cup \cdots \cup v_n$ é a view de fecho de t referente ao caminho maximal p. O conjunto de views de fecho de um thread t é denotado por C_t.*

A operação fecho de um programa dá-nos um novo programa que contém novos *threads* adicionais. As *views* destes *threads* adicionais representam cenários de acessos que não ocorrem na versão atual do programa. No entanto, porque existe uma correlação entre as *views* do programa original dada pelo predicado de dependência δ, estes novos *threads* representam cenários de execução que poderão vir a ser introduzidos em versões futuras e que potencialmente conduzirão a HLDR.

Definição 4 (Fecho de um Programa) *A operação de fecho de um programa P, $clos(P)$, é um novo programa que consiste no programa P ao qual foi adicionando um novo thread t_v para cada view v em $\bigcup_{t \in T} C_t$ tal que $V(t_v) = \{v\}$, onde T é o conjunto de threads de P.*

Este *threads* adicionados pela operação de fecho têm sempre uma e uma só *view*, pelo que as interações entre estes novos *threads* nunca introduz novas HLDR.

Definição 5 (Programa Fechado) *Um programa P está fechado se as anomalias detetadas em P são as mesma que as detetadas em $clos(P)$.*

O conjunto de HLDR num programa fechado mantém-se fixo e estável em versões futuras do mesmo programa. Se um programa fechado não tiver HLDR, então é garantido que versões futuras do mesmo programa não irão sofrer de HLDR.

3.1 Exemplos

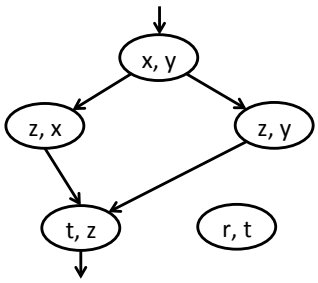
Nos exemplos apresentados a construção **`atomic`** representa uma transação em memória.

Exemplo 1

Thread t

```
atomic { x = y; }
if (cond1)
    atomic { z = x; }
else
    atomic { z = y; }
if (cond2)
    atomic { t = z; }
else
    atomic { r = t; }
```

Grafo de dependências G_t



$$C_t = \{\{x, y, z, t\}, \{x, y, z, t\}, \{r, t\}\}$$
$$= \{\{x, y, z, t\}, \{r, t\}\}$$

Consideremos o programa P_1 cujo único *thread* t que executa o código acima à esquerda. À direita do programa está o seu grafo de dependências de acordo com o predicado $\delta(u, v)$ definido anteriormente. As *views* de fecho de P_1 são $\{x, y, z, t\}$ e $\{r, t\}$. Esta última *view* de fecho foi criada a partir de um caminho maximal de comprimento unitário e, como já existe em P_1, irá ser ignorada como *view* de fecho.

Portanto o fecho do programa P_1, denotado por $clos(P_1)$, consiste em adicionar a P_1 um novo *thread* t' com a *view* $\{x, y, z, t\}$.

$clos(P_1) =$	t	$\cup$	t'
	$\{x, y\}$		$\{x, y, z, t\}$
	$\{z, x\}$		
	$\{t, z\}$		
	$\{z, y\}$		
	$\{r, t\}$		

É de realçar que existe uma relação de dependência dada pelo controlo de fluxo do programa entre a *view* $\{r, t\}$ e outras *view* de t. No entanto, se aplicarmos a relação de dependência δ tal como definida anteriormente, a *view* $\{r, t\}$ é independentes das demais *views* de t.

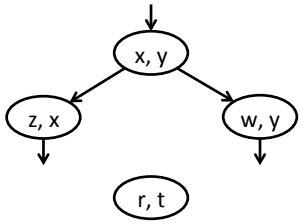
Este exemplo mostra como a operação de fecho captura potenciais novas HLDR que não existiam no programa original, que aliás era *single-threaded*, através da introdução de novas *views* maximais.

Exemplo 2

Thread t

```
atomic { x = y; }
if (cond1)
    atomic { z = x; }
else
    atomic { w = y; }
atomic { r = t; }
```

Grafo de dependências G_t



$C_t = \{\{x, y, z\}, \{x, y, w\}, \{r, t\}\}$

Consideremos o programa P_2 com um único *thread* t que executa o código apresentado em cima à esquerda, a que corresponde o grafo de dependências apresentado à sua direita. O fecho do programa P_2, $clos(P_2)$, vai conter dois novos *threads* t' e t'' com as *views* $\{x, y, z\}$ e $\{x, y, w\}$ respetivamente.

$clos(P_2) =$	t	$\cup$	t'	$\cup$	t''
	$v_1 = \{x, y\}$		$v' = \{x, y, z\}$		$v'' = \{x, y, w\}$
	$v_2 = \{z, x\}$				
	$v_3 = \{w, y\}$				
	$v_4 = \{r, t\}$				

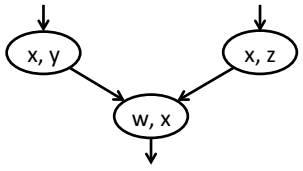
O novo programa $clos(P_2)$ tem agora duas HLDR uma obtida pelas *views* v_1 e v_2 com v', e outra obtida pelas *views* v_1 e v_3 com v''. Cada *thread* adicionado no processo de fecho do programa representa um conjunto acessos às variáveis partilhadas que, embora não ocorrendo no programa original, podem vir a ocorrer em futuras versões do programa, introduzindo assim HLDR que, por envolverem trechos de código pré-existentes à nova versão, serão particularmente difíceis de identificar.

Exemplo 3

Thread t_i

```
if (cond)
    atomic { x = y; }
else
    atomic { x = z; }
atomic { w = x; }
```

Grafo de dependências G_{t_i}



$C_t = \{\{w, x, y\}, \{w, x, z\}\}$

Consideremos o programa P_3 com dois *threads* que executam concorrentemente o trecho de programa ilustrado em cima à esquerda, mas onde `cond` é falso para um dos *threads* e verdadeiro para o outro. A este programa corresponde o grafo de dependências em cima à direita. Este programa não tem HLDR (ver Definição 1).

$clos(P_3) =$	t_1	$\cup$	t_2	$\cup$	t'	$\cup$	t''
	$v_1^1 = \{x, y\}$ $v_2^1 = \{w, x\}$		$v_1^2 = \{x, z\}$ $v_2^2 = \{w, x\}$		$v' = \{w, x, y\}$		$v'' = \{w, x, z\}$

Como os dois *threads* em P_3 executam trechos de código diferentes, os seus conjuntos de *views* são também diferentes, tal como ilustrado na tabela. O fecho de P_3 irá adicionar dois novos *thread* t' e t'' com as *views* $v' = \{w, x, y\}$ e $v' = \{w, x, z\}$, respetivamente.

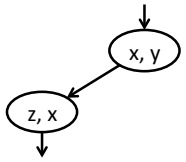
Este exemplo ilustra o caso onde um grafo de dependências tem mais que um "ponto de entrada", neste caso há dois pontos de entrada no grafo, a que correspondem dois caminhos maximais diferentes.

Exemplo 4

Thread t_i

```
atomic { x = y; }
if (cond)
    atomic { z = x; }
```

Grafo de dependências G_{t_i}



$C_t = \{\{x, y, z\}\}$

Consideremos o programa P_4 com múltiplos *threads* que executam concorrentemente o trecho de programa ilustrado em cima à esquerda. Este programa não tem HLDR (ver Definição 1).

Como todo os *threads* em P_4 executam o mesmo código, os seus conjuntos de *views* são idênticos $V(t_i) = \{v_1, v_2\}$ onde $v_1 = \{x, y\}$ e $v_2 = \{x, z\}$, pelo que todos têm também grafos de dependências G_{t_i} idênticos. Portanto, o fecho de P_4 irá adicionar apenas um novo *thread* t' com a *view* $v' = \{x, y, z\}$.

O novo programa $clos(P_4)$ tem uma HLDR visto que $v_1 \cap v' = v_1$ e $v_2 \cap v' = v_2$, portanto $\varphi(t_i, v') = \{v_1, v_2\}$ não é totalmente ordenado sobre $\subseteq$.

Exemplo 5

Thread t_1

```
atomic { x = y; }
atomic { z = x; }
```

Thread t_2

```
atomic { x = y + z; }
```

Consideremos o program P_5 com dois *threads* t_1 e t_2 executand os trechos de código acima. Neste exemplo $clos(P_5)$ iria adicionar um novo *thread* com a *view* $v' = \{x, y, z\}$, que já existem em P_5 através do *thread* t_2. Logo, as HLDR que possam existir em $clos(P_5)$ também existem em P_5, portanto P_5 é um programa fechado.

4 Metodologia para Tornar uma Classe *Thread-Safe*

Nesta secção vamos mostrar como se pode usar a operação de fecho de programas para identificar anomalias que podem resultar da invocação concorrente de métodos de uma classe.

Queremos tornar uma classe *thread-safe*, i.e. assegurar que qualquer entrelaçamento de qualquer sequência de chamadas aos seus métodos é livre de HLDRs. Desta análise vai-se poder determinar quais os métodos que devem ser chamados atomicamente. O programador da classe deve então documentar devidamente estas possíveis utilizações incorretas ou redesenhar a API de forma a evitar-las.

Consideramos que cada método da API da classe pode ser chamado concorrentemente e pode criar novos *threads*. Seja então a API composta pelos métodos públicos $m_1, \cdots, m_n$. Modelamos cada método m_i como um programa P_i e definimos um novo programa P_{API} como o programa que consiste na execução concorrente dos programas $P_1, \cdots, P_n$. Intuitivamente P_{API} representa todas as interações possíveis em cenários de execução concorrente dos métodos da API da classe.

Vamos mostrar que todas as anomalias em P_{API} podem ser expostas analisando apenas os fechos de cada método individualmente, permitindo assim desta forma analisar eficientemente as anomalias de P_{API}.

Teorema 1 *Todas as high level data races detectadas em P_{API}, ou existem em $clos(P_i)$ para algum i, ou então são falsos positivos.*

Demonstração. Seja α uma HLDR em P_{API} entre o *thread* t_I e t originado por $v_m \in t_I$ e t é um *thread* de P_i. Sabemos então, que pela Definição 1, $\varphi(t, v_m)$ não é uma cadeia, ou seja, existe $v, v' \in V(t)$ tal que $v \cap v_m \not\subseteq v' \cap v_m \wedge v \cap v_m \not\supseteq v' \cap v_m$ o que implica

$$v \not\subseteq v' \ \wedge \ v \not\supseteq v'. \tag{1}$$

Então existem duas possibilidades: i) v e v' pertencem ao mesmo caminho maximal em G_t; ou ii) v e v' pertencem a caminhos maximais diferentes.

i) Se v e v' pertencem ao mesmo caminho maximal em G_t então, pela Definição 4, $clos(P_i)$ contém um novo *thread* t' com uma *view* ν tal que $v \cup v' \subseteq \nu$. Da combinação desta condição com (1) pode-se inferir que existe uma HLDR correspondente a α entre os *threads* t' e t, visto que $\varphi(t, \nu) = \{v \cap \nu, v' \cap \nu, \cdots\}$ não forma uma cadeira pois $v \cap \nu \not\subseteq v' \cap \nu \ \wedge \ v \cap \nu \not\supseteq v' \cap \nu$.
ii) Se v e v' pertencem a caminhos maximais distintos em G_t, então pela definição do predicado de dependência δ, ou não há relação de controlo de fluxo entre as *views*, i.e. $v \not\prec v' \ \wedge \ v \not\succ v'$, ou as *views* v e v' são completamente disjuntas, i.e., $v \cap v' = \emptyset$. Em ambos estes casos, a anomalia corresponde a um falso positivo em P_{API}. □

Note-se que esta prova não assume que $t \neq t_I$, pelo que se considera o caso em que um método da API é executado em concorrência com ele próprio.

O segundo caso da prova do Teorema 1 lida com o caso em que as *views* v e v' não pertencem ao mesmo caminho máximo de G_t, e por isso a anomalia detetada é um falso positivo. O seguinte exemplo ilustra essa situação:

Thread t

```
if (cond)
  atomic { x = y; }
else
  atomic { z = x; }
```

Thread t_I

```
atomic { x = y + z; }
```

Neste exemplo temos que a *view* maximal de t_I é $v_m = \{x, y, z\}$ e as *views* de $V(t) = \{\{x, y\}, \{z, x\}\}$, logo $\varphi(t, v_m) = \{\{x, y\}, \{x, z\}\}$ não forma uma cadeia e, portanto, é identificada uma HLDR que depende da execução de ambas as transações. No entanto, esta HLDR representa um falso positivo, pois as *views* $\{x, y\}$ e $\{x, z\}$ são exclusivas, pelo que apenas uma e uma só das transações será executada, nunca ambas.

5 Resultados Experimentais

A analise apresentada neste artigo foi implementa na *framework MoTH* [5], uma ferramenta de verificação de HLDR que analisa código transacional em *Java Bytecode*. Esta versão estendida da *framework MoTH* foi aplicada para analisar automaticamente um programa exemplo baseado numa variante do problema clássico de manipulação de contas bancárias.

Este programa baseia-se numa classe que gere uma conta bancária. A classe associa à conta o saldo disponível, um registo de todos os depósitos feitos e um contador de operações efetuadas, sejam elas depósitos, levantamentos, transferências bancarias, ou outras. Em particular, a operação de `deposit` obtém o saldo atual da conta, atualiza-o localmente e depois regista o novo saldo, tudo isto recorrendo aos métodos públicos (API) da classe `getBalance` e `setBalance` que são executados como transações em memória. Depois de atualizado o saldo, a operação são invocados os métodos públicos `saveDepLog` e `incCountOper` para registar a operação no histórico de operações e incrementado o contador de operações, respetivamente. Todos estes métodos são atómicos e executados como transações em memória, à exceção da operação de depósito. No entanto, todos os acessos aos campos da classe realizados na operação de `deposit` são realizados através dos métodos atómicos atrás referidos. Este programa não contem HLDR segundo a caracterização da Definição 1.

O fecho deste programa introduz um novo *thread* que acede ao saldo, ao histórico e ao contador de operações, que corresponde ao facto do método que regista a operação de depósito no histórico poder ler um valor de saldo não relacionado com o depósito. Este método de registo de depósito é publico, e por isso compromete a *thread-safety* da classe, tal como detetado pela implementação de fecho do programa e posterior análise de HLDR realizado na versão estendida da *framework MoTh*.

6 Trabalho Relacionado

Existem vários trabalhos que lidam com o problema da deteção de violações de atomicidade, que usam técnicas de análise estática ou dinâmica.

Artho et al. [1] introduziu a noção de *view* de blocos atómicos que representa o seu acesso a variáveis partilhadas, bem como a noção de *views* maximais de um *thread*, que são usados para definir *consistência de views*, que quando violada, indica uma potencial *high level data race*. Este trabalho é construído sobre estes conceitos, mas discute conceitos adicionais como o grafo de dependências entre *views* e o fecho de um programa.

Shacham et al. [6] propõe uma metodologia para testar a atomicidade da composição de operações concorrentes. A técnica proposta é baseada no teste de código cliente na presença de um adversário e usa especificações sobre a comutatividade das operações para reduzir o número de execuções tidas em conta para detetar uma violação de atomicidade. A nossa aproximação lida com a análise de uma API sem código cliente e reduz bastante o espaço de procura de anomalias por aplicar a operação de fecho e analisar cada método da API em separado.

Farchi et al. [2] introduz o conceito de fecho de um programa e a sua aplicação para tornar uma biblioteca *thread-safe*. Este artigo, dos mesmos autores, refina aquele trabalho focando-o no paradigma da memória transacional e da programação em Java com orientação a objetos, introduzindo também novos exemplos e uma reformulação completa do Teorema 1. Este artigo informa sobre a extensão da *framework MoTH* para incluir o conceito de fecho de programa e reporta os resultados da aplicação desta *framework* automática sobre um exemplo de um caso prático.

7 Conclusões

Neste artigo discutimos como transformar uma classe Java de forma a torná-la *thread-safe* e as suas instâncias poderem ser usadas num contexto *multi-threading*. Para isso apresenta-se uma metodologia baseada no conceito inovador de *fecho de programa*, que permite identificar as *high level data races* existentes na implementação dos métodos de uma classe (caso usem múltiplos *threads* nessa implementação), bem como as resultantes da execução concorrente desses métodos. Estes conceitos são ilustrados recorrendo a um conjunto de exemplos. A metodologia descrita neste artigo é totalmente focada na classe a transformar e não depende em nada da aplicação que está a classe transformada. Foi realizada uma extensão da *framework MoTH* para incluir o conceito de fecho de um programa e detetar as HLDR no programa resultante. Esta nova versão da *framework* foi aplicada numa variante do exemplo clássico da manipulação de contas bancárias, demonstrando a utilidade do conceito de fecho de programa e a sua aplicabilidade na identificação de restrições à utilização concorrente de métodos públicos de uma classe Java.

Referências

1. Cyrille Artho, Klaus Havelund, and Armin Biere. High-level data races. *Software Testing, Verification and Reliability*, 13(4):207–227, December 2003.
2. Eitan Farchi, Segall Itai, João M. Lourenço, and Diogo Sousa. Using program closures to make an application programming interface (API) implementation thread safe. In *Proceedings of the Workshop on Parallel and Distributed Systems: Testing, Analysis, and Debugging*, PADTAD'12, New York, NY, USA, 2012. ACM. To appear.
3. Yang Ni, Adam Welc, Ali-Reza Adl-Tabatabai, Moshe Bach, Sion Berkowits, James Cownie, Robert Geva, Sergey Kozhukow, Ravi Narayanaswamy, Jeffrey Olivier, Serguei Preis, Bratin Saha, Ady Tal, and Xinmin Tian. Design and implementation of transactional constructs for c/c++. *SIGPLAN Not.*, 43(10):195–212, October 2008.
4. Vasco Pessanha. Verificação prática de anomalias em programas de memória transaccional, September 2011.
5. Vasco Pessanha, Ricardo J. Dias, João M. Lourenço, Eitan Farchi, and Diogo Sousa. Practical verification of high-level dataraces in transactional memory programs. In *Proceedings of the Workshop on Parallel and Distributed Systems: Testing, Analysis, and Debugging*, PADTAD'11, pages 26–34, New York, NY, USA, 2011. ACM.
6. Ohad Shacham, Nathan Bronson, Alex Aiken, Mooly Sagiv, Martin Vechev, and Eran Yahav. Testing atomicity of composed concurrent operations. In *Proceedings of the 2011 ACM international conference on Object oriented programming systems languages and applications*, OOPSLA '11, pages 51–64, New York, NY, USA, 2011. ACM.